网络空间安全系列教材
普通高等教育“十三五”规划教材

典型密码算法 FPGA 实现

杨亚涛　李子臣　编著

電子工業出版社
Publishing House of Electronics Industry
北京 · BEIJING

内 容 简 介

本书用 FPGA 实现的密码算法主要分为四大部分，分别是分组密码、公钥密码、Hash 算法和数字签名算法，其中分组密码包括 DES、AES 和 SM4 算法；公钥算法包括 RSA 公钥密码算法、ECC 密码算法和 SM2 密码算法；Hash 算法包括 SHA-1 算法、SHA-3 算法和 SM3 算法；数字签名算法包括 ECC 签名算法和 DSA 签名算法。

本书在 Xilinx 公司的 ISE 平台和 Mentor 公司 ModelSim 仿真软件上编程实现了这些算法，并且还附加了相关实现截图以及密码算法实现效率分析。

本书不仅可作为大学密码与信息安全相关专业本科生以及研究生的教学与参考用书，也可以作为密码与信息安全科研或工程开发人员的参考书。

图书在版编目（CIP）数据

典型密码算法 FPGA 实现 / 杨亚涛，李子臣编著．—北京：电子工业出版社，2017.1
ISBN 978-7-121-30383-8

I. ①典… II. ①杨…②李… III. ①密码算法－可编程序逻辑器件－系统设计 IV. ①TN918.1②TP332.1

中国版本图书馆 CIP 数据核字（2016）第 277861 号

策划编辑：戴晨辰
责任编辑：戴晨辰　　文字编辑：刘　芳
印　　刷：
装　　订：
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　　邮编：100036
开　　本：787×1 092　1/16　印张：13　字数：332.8 千字
版　　次：2017 年 1 月第 1 版
印　　次：2017 年 1 月第 1 次印刷
定　　价：38.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式：dcc@phei.com.cn，192910558（QQ 群）。

网络空间安全系列教材

编委会名单

序

随着经济全球化和信息化的发展，以互联网为平台的信息基础设施，对整个社会的正常运行和发展正起着关键的作用。甚至，像电力、能源、交通等传统基础设施的运行，也逐渐依赖互联网和相关的信息系统才能正常运行。网络信息对社会发展有重要的支撑作用。

网络空间是利用全球互联网和计算系统进行通信、控制和信息共享的动态虚拟空间，包括四个要素，分别是网络平台、用户虚拟角色、资产数据和管理活动，是社会有机运行的神经系统，已经成为继陆、海、空、天之后的第五空间。

网络空间面临的威胁也与日俱增。从国际上看，国家或地区在政治、经济、军事等各领域的冲突都会反映到网络空间中，而由于网络空间边界不明确、资源分配不均衡，导致网络空间的争夺异常复杂。另外，网络犯罪和网络攻击也对个人和企业构成严重威胁。在网络中，个人隐私信息泄露并大范围传播的事件已经屡见不鲜，以非法牟利为目的、利用计算机网络进行的犯罪已经形成了黑色的地下经济产业链。如何充分利用互联网对经济发展的推动作用、保护公民和企业的合法权益，同时又要控制其对经济社会发展带来的负面威胁，需要研究和探索更加科学合理的网络空间安全治理模式。正如习近平总书记所言：“没有网络安全，就没有国家安全”。

加强网络空间安全已经成为国家安全战略的重要组成部分。2014 年 2 月，中央网络安全和信息化领导小组成立。2015 年 6 月，国务院学位委员会、教育部决定在“工学”门类下增设“网络空间安全”一级学科，并明确指出需加强“网络空间安全”的学科建设，做好人才培养工作。2016 年 3 月，国务院学位委员会下发通知，明确全国共有 29 所高校获得我国首批网络空间安全一级学科博士学位授权点。6 月，中央网络安全和信息化领导小组办公室、国家发展和改革委员会、教育部、科学技术部、工业和信息化部、人力资源和社会保障部联合发文，《关于加强网络安全学科建设和人才培养的意见》（中网办发文[2016]4 号）指出，网络空间的竞争，归根结底是人才竞争。我国网络空间安全人才还存在数量缺口较大、能力素质不高、结构不尽合理等问题，与维护国家网络安全、建设网络强国的要求不相适应。提出要加快网络安全学科专业和院系建设；创新网络安全人才培养机制；加强网络安全教材建设；强化网络安全师资队伍建设；完善网络安全人才培养配套措施等意见。

网络空间安全主要研究网络空间中的安全威胁和防护问题，即在有敌手的对抗环境下，研究信息在产生、传输、存储、处理、销毁等各个环节中所面临的威胁和防御措施，以及网络和系统本身面临的安全漏洞和防护机制，不仅仅包括传统信息安全所研究的信息的保密性、完整性和可用性，同时还包括构成网络空间基础设施的安全和可信。从宏观层面来看，网络空间安全的研究对象主要包括：全球各类各级信息基础设施的安全威胁；从微观来看，主要对象包括：通信网络、计算机网络及其设备和应用系统中的安全威胁。

数学、信息论、计算复杂性理论等是网络空间安全所依靠的重要理论基础。

网络空间安全的理论体系由三部分组成。一是基础理论体系，主要包括：网络空间理论、密码学、离散结构理论和计算复杂性理论等；其中，信息的机密性、完整性、可控性、可靠

性等是核心，对称加密、公钥加密、密码分析、侧信道分析等是重点，在复杂环境中的可证安全、可信可控及定量分析理论是关键。二是技术理论体系，主要包括网络空间安全保障理论体系，从系统和网络角度，研究和设计网络空间的各种安全保护方法和技术。重点包括：芯片安全、操作系统安全、数据库安全、中间件安全、恶意代码等，从预警、保护、检测到恢复响应的安全保障技术理论。从网络安全角度，以通信基础设施、互联网基础设施等为研究对象，聚焦研究通信安全、网络安全、网络对抗等。三是应用理论体系，从应用角度来看，针对各种应用系统，研究在实际环境中面临的各种安全问题，如 Web 安全、内容安全、垃圾信息等，涵盖电子商务、电子政务、物联网、云计算、大数据等诸多应用领域。

网络空间安全有如下五个研究方向。一是网络空间安全基础，包括：网络空间安全数学理论、网络空间安全体系结构、网络空间安全数据分析、网络空间博弈理论、网络空间安全治理与策略、网络空间安全标准与评测等。二是密码学及应用，包括：对称密码设计与分析、公钥密码设计与分析、安全协议设计与分析、侧信道分析与防护、量子密码与新型密码等。三是系统安全，包括：芯片安全、系统软件安全、虚拟化计算平台安全、恶意代码分析与防护等。四是网络安全，包括：通信基础设施及物理环境安全、互联网基础设施安全、网络安全管理、网络安全防护与主动防御（攻防与对抗）、端到端的安全通信等。五是应用安全，包括：关键应用系统安全、社会网络安全（包括内容安全）、隐私保护、工控系统与物联网安全、先进计算安全等。

中国密码学会教育与科普工作委员会与电子工业出版社合作，共同筹划了这套“网络空间安全系列教材”，主要包括《密码学》、《密码学实验教程》、《公钥密码学》、《应用密码学》、《密码学数学基础》、《密码基础算法》、《典型密码算法 FPGA 实现》、《典型密码算法 JAVA 实现》、《公钥密码算法 C 语言实现》、《密码分析学》、《网络空间安全导论》、《信息安全管理》、《信息系统安全》、《网络空间安全技术》、《网络空间安全实验教程》、《网络攻防技术》、《同态密码学》、《对称密码学》等。希望为信息安全、网络空间安全、网络安全与执法、信息对抗技术等本科专业提供教材，也为密码学、网络空间安全、信息安全等专业的研究生和博士生，以及从事该领域的科研人员提供教材和参考书。为我国网络空间安全教材建设、普及密码知识和网络空间安全人才培养，贡献绵薄之力。

2016 年 12 月

前　言

信息在社会中的地位和作用越来越重要，已成为社会发展的重要战略资源，随之而产生的信息安全问题也日益受到人们的关注，保证信息的安全是保障国家稳定、促进经济发展的重要因素。为了提高我国信息安全的建设水平，提升大学生对密码学与信息安全相关知识的掌握程度和运用能力，国内许多高校开设了不少有关密码学和信息安全的课程，但是所用教材与参考书籍大多侧重于密码算法理论与原理的描述与分析，缺乏对算法的实现过程与实现环境的具体描述，对算法代码的硬件实现更少提及。许多学生学习起来感觉比较茫然和枯燥，以致最后对密码算法的掌握不够深入扎实，对密码学相关知识的学习效果不够理想。因此，本着帮助读者学习、研究密码算法的初衷，本书主要描述典型密码算法的 FPGA 实现过程，侧重培养读者的编程能力，在前人工作的基础上，根据国家公布的有关标准密码算法以及密码学研究的热点，就现行的主要密码算法进行了编程实现。

本书内容丰富、特色鲜明、实用性强，不仅给出了算法的理论知识，还在 Xilinx 公司的 ISE 平台和 Mentor 公司 ModelSim 仿真软件上编程实现了整个算法，并且还附加了相关实现截图以及密码算法实现效率分析。本书不仅可以作为大学本科生以及研究生的教学与参考用书，也可以作为密码科学研究者与工程开发人员的参考书。

本书密码算法主要分四大部分，分别是分组密码、公钥密码、Hash 算法和数字签名算法，其中分组密码有 DES、AES 和 SM4 算法，公钥算法有 RSA 公钥密码算法、ECC 密码算法和 SM2 密码算法，Hash 算法有 SHA-1 算法、SHA-3 算法和我国商密算法 SM3，数字签名算法有 ECC 签名算法和 DSA 签名算法。

本书各章程序实现的参考源代码可以通过华信教育资源网 http://www.hxedu.com.cn 注册免费下载。

全书由杨亚涛博士、李子臣教授负责编著，本书的编写得到了北京电子科技学院相关领导和师生的无私帮助，在此向所有为本书做出贡献的老师和同学们致以衷心的感谢！电子工业出版社为本书的校对、编辑和出版做了大量的工作，对他们也表示诚挚的感谢！

由于时间仓促以及作者水平有限，虽然尽全力对本书进行了校对和检错，但是不免还有疏漏之处，恳请广大读者批评指正。

作　者

2016 年 12 月

目　录

第 1 章　密码算法 FPGA 实现基础

1.1 FPGA 概述

目前市场上的 FPGA 芯片主要来自 Xilinx 公司和 Altera 公司，这两家公司占据了 FPGA 80%以上的市场份额，是 FPGA 的主流厂商。除此之外，还有 Actel、Lattice、Atmel 等公司生产相关功能的 FPGA 芯片。

Xilinx（赛灵思）是全球领先的可编程逻辑完整解决方案的供应商，它研发、制造并销售成系列的高级集成电路、软件设计工具以及作为预定义系统级功能的 IP（Intellectual Property）核。客户使用 Xilinx 及其合作伙伴的自动化软件工具和 IP 核对器件进行编程，从而完成特定的逻辑操作。Xilinx 公司成立于 1984 年，首创了现场可编程逻辑阵列（Field Programmable Gate Array，FPGA）这一创新性的技术，并于 1985 年首次推出商业化产品。目前 Xilinx 满足了全世界对 FPGA 产品一半以上的需求，Xilinx 的主流 FPGA 分为两大类：一种侧重低成本应用，容量中等，性能可以满足一般的逻辑设计要求，如 Spartan 系列；还有一种侧重于高性能应用，容量大，性能能满足各类高端应用，如 Virtex 系列。用户可以根据自己实际应用的需求进行选择，在性能可以满足的情况下，优先选择低成本器件。

自 20 世纪 80 年代发明世界上第一个可编程逻辑器件开始，Altera 公司一直秉承着创新的传统，成为世界上“可编程芯片系统”（System on a Programmable Chip，SOPC）解决方案倡导者。Altera 结合带有软件工具的可编程逻辑技术、知识产权（IP）和技术服务，在世界范围内为 14000 多个客户提供过高质量的可编程解决方案。产品系列将可编程逻辑的内在优势（灵活性）、产品及时升级和集成化结合在一起，可以满足客户的不同需求。

Actel 公司成立于 1985 年，位于美国纽约。之前的 20 多年里，Actel 一直效力于美国军工和航空，且被禁止对外出售产品。后来开始逐渐转向民用和商用，出售的产品除了反熔丝系列外，还推出了可重复擦除的 ProASIC3 系列。

Lattice 半导体公司也提供业界成系列的现场可编程门阵列（FPGA）、可编程逻辑器件（Programmable Logic Device，PLD）及其相关软件，包括现场可编程系统芯片（Field Programmable System Chip，FPSC）、复杂的可编程逻辑器件（Complex Programmable Logic Device，CPLD），可编程混合信号产品和可编程数字互连器件。FPGA 和 PLD 是广泛使用的半导体元件，最终用户可以将其配置成特定的逻辑电路，从而缩短设计周期，降低开发成本。

Atmel 公司是世界上高级半导体产品设计、制造和营销的领先者，推出的产品包括微处理器、可编程逻辑器件、非易失性存储器、安全芯片、混合信号及 RF 射频集成电路等。

1.1.1 Xilinx 公司的代表芯片

1．面向高性能的 Virtex-5 FPGA 系列

系统集成平台——Virtex-5 系列 FPGA 提供了 4 种新型平台，每种平台都在高性能逻辑、串行连接、信号处理和嵌入式处理性能方面实现了最佳平衡。

常用的 3 款平台特性如下。

（1）Virtex-5 LX 平台：针对高性能逻辑进行了优化。

（2）Virtex-5 LXT 平台：针对带有低功耗串行连接功能的高性能逻辑进行了优化。

（3）Virtex-5 SXT 平台：针对带有低功耗串行连接功能的 DSP 和存储器密集型应用进行了优化。

2．面向低成本的 Spartan-3 FPGA 系列

90nm Spartan-3 系列 FPGA 的发售量已经超过 3 000 万片，它是业内首款大容量 FPGA 系列产品，针对多个特定领域进行了平台优化。

（1）面向数字信号处理的 Spartan-3A DSP 平台。这个平台对 DSP 进行了优化，适合那些需要集成 DSP MAC 和扩展存储器的应用，特别适合那些需要低成本 FPGA 来实现信号处理（如军用无线电、监视照相机、医学成像等）的应用设计。

（2）面向非易失性应用的 Spartan-3AN 平台。这个平台主要针对非易失性、系统集成、安全、大型用户 Flash 的应用，特别适用于低成本嵌入式控制器。

（3）面向主流应用的 Spartan-3 平台。

① Spartan-3A 平台：针对 I/O 进行了优化。这个平台主要应用在对 I/O 数目和性能比要求较高的场合，特别适于桥接、差分信号和存储器接口等需要宽接口或者多个接口以及一定处理能力的应用。

② Spartan-3E 平台：针对逻辑进行了优化。这个平台主要应用在对逻辑密度要求较高的场合，特别适于逻辑集成、DSP 协处理和嵌入式控制等这些需要进行大量处理和窄接口或者少量接口的应用。

③ Spartan-3 平台：这个平台主要针对那些高逻辑密度和高 I/O 数目的应用，特别适用于高度集成的数据处理应用。

1.1.2 Altera 公司的代表芯片

1．面向高性能的 Stratix III FPGA 系列

与 Xilinx 的 Virtex-4 系列对应，Altera 公司推出了 Stratix III 系列 FPGA 体系结构。Stratix III 系列不仅性能比上一代提高很多，更重要的是静态和动态功耗比前代 FPGA 降低了 50%。Stratix III 器件经过设计，支持高速内核以及高速 I/O，并且具有非常好的信号完整性，例如，它能够实现 400MHz DDR3 的 FPGA。这种性能的提高源于以下几点：增强的 DSP 模块实现了信号处理算法；优化的内部存储器改进了信号存储器接口；高性能的外部存储器接口改进了布线体系结构；灵活的 I/O 支持最新的外部存储器标准。为了给客户的设计应用提供最高性价比的解决方案，Altera Stratix III FPGA 提供了 3 种型号，分别针对逻辑、DSP 和存储器以及收发器进行了优化。

2．面向低成本的 Cyclone III FPGA 系列

低成本的 Cyclone III FPGA 是 Altera Cyclone 系列的第三代产品。Cyclone III FPGA 系列同时实现了低功耗、低成本和高性能，进一步扩展了 FPGA 的应用；Cyclone III FPGA 采用 TSMC 公司的 65-nm 低功耗（LP）工艺技术，Cyclone III 器件对芯片和软件采取了更多的优化措施，在所有 65-nm FPGA 中是功耗最低的，在对成本和功耗敏感的大量应用中，显示出很大的优势。Cyclone III 系列包括 8 个型号，具有 5k 到 120k 个逻辑单元（LE），最多有 534 个 I/O 引脚。Cyclone III 器件具有 4MB 嵌入式存储器、288 个嵌入式 18×18 乘法器、专用外部存储器接口电路、锁相环（PLL）以及高速差分 I/O 等模块。

1.2 FPGA 工作原理

FPGA 是基于 PAL（Programmable Array Logic）、GAL（General Array Logic）、CPLD 等开发出来的新技术，常作为专用集成电路控制的分电路，如图 1.1 所示，在原有可编程控制器中进行了优化调整。FPGA 采用了逻辑单元阵列这样一个概念，内部包括可配置逻辑模块、输出输入模块和内部连线三个部分。FPGA 利用小型查找表（16x1RAM）来实现组合逻辑，每个查找表连接到一个 D 触发器的输入端，触发器再来驱动其他逻辑电路或驱动 I/O，由此构成了既可实现组合逻辑功能又可实现时序逻辑功能的基本逻辑单元模块。

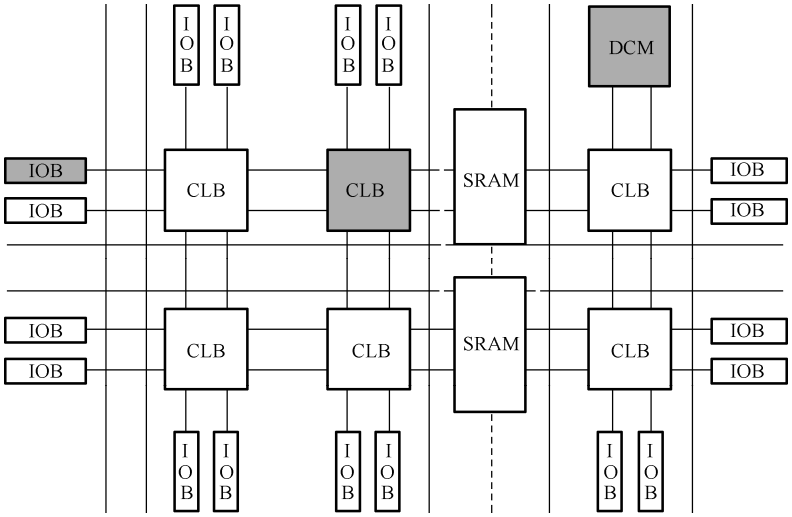

图 1.1　FPGA 内部结构

FPGA 结构模块的功能描述如下。

BRAM（Bipolar Random Access Memory）模块：专用集成电路是服务于各个行业控制的应用型电路，FPGA 在结构模块布局方而也要适应实际操作的要求。一般情况下，FPGA 配备了专用的内嵌式随机存储器（RAM）来针对数据端口的传输位置、存储结构、元件功能等要素，提供一个稳定的逻辑存储方式。以内存储操作为例，FPGA 搜索到某个触发器时，要准确地感应出电路中动态或静态存储方式，再由 BRAM 辅助存储器作为电路信号收录装置。

DCM（Data Control Memory）模块：FPGA 的逻辑是通过向内部静态存储单元加载编程

数据来实现的，业内大多数 FPGA 均提供数字时钟管理。先进的 FPGA 不仅提供数字时钟管理，还提供相位环路锁定，相位环路锁定能够提供精确的时钟综合，并实现过滤功能。数字控制模块（DCM）的主要作用是对门电路各程序指令进行规划，限定数据信号传输的标准位置，避免数据冗杂从而提升电子系统控制的效率。另外，数字控制模块也对门电路运行时间进行精确控制，防止信号延迟而造成电子仪器的误动作。

CLB（Configurable Logic Block）模块：逻辑处理是 FPGA 中处理数据的有序流程，按照电路信号编码程序的规则对门电路进行优化编程。可配置逻辑块（CLB）的实际数量和特性会根据器件的不同而不同，但每个 CLB 都包含一个可配置的开关矩阵，此矩阵由 4 或 6 个输入以及一些选型电路和触发器组成。

IOB（Input Output Block）模块：可编程输入输出单元（IOB）是现场可编程门阵列的基本构造，输入/输出（I/O）模块负责 FPGA 数据信号收录、传输作业要求等。从结构层次来划分，IOB 模块是芯片与外界电路的接口部分，完成不同电气特性下对输入/输出信号的驱动与匹配要求。I/O 接口由多个单元组成，按照电路的相位、电阻、元控件等指标，严格控制门电路的运作流程。

总之，FPGA 是基于 SRAM 的可编程器件，它以功能很强的 CLB 为基本逻辑单元，可以实现各种复杂的逻辑功能，FPGA 还具有可扩展的优点。由于 FPGA 的性能和灵活性，以及新的简明设计和实施方法，在很多新兴应用领域中 FPGA 都成为优选的解决方案。

1.3 FPGA 语法基础

Verilog HDL 和 VHDL 是目前最流行的硬件描述语言，均为 IEEE 标准，被广泛应用于可编程逻辑器件的项目开发。两者都是用于逻辑设计的硬件描述语言，都能形象化地表示电路的行为和结构，支持逻辑设计中层次与范围的描述，可以描述简化电路的行为，具有电路仿真验证机制，支持电路描述由高层到低层的转换，便于管理与设计混用。

但两者又有不同的特点：Verilog HDL 产生较早，容易被接受，学习者如果有 C 语言基础，就能很快掌握它；但是它在系统抽象方面比较弱，不适合大型系统的开发，经过不断的升级，其系统级表述性能和可综合性能得到了大幅度提升。VHDL 需要 Ada 编程语言基础，学习者需要较长时间才能全面掌握它。不过，这两种语言仍处于不断完善当中，都在向更高级描述语言的方向迈进。

1.3.1 Verilog HDL 语法要点

1．一般的模块结构

module<模块名>（<端口列表>）
<定义>
<模块条目>
endmodule

2．IO 端口种类

Input 表示模块从外界读取数据的接口，在模块内不可写。Output 表示模块往外界送出数据的接口，在模块内不可读。Inout：可读取数据，也可以送出数据。

对相同位宽的输入输出信号可以一起声明，比如，input[3:0] a,b；不同位宽的必须分开写，比如，input [5:0] a; input [6:0] b；内部信号为 reg 类型，内部信号的状态有 0、1、x、z。

3．赋值运算符

逻辑功能描述中，常用 assign 语句来描述组合逻辑电路，always 语句既可以描述组合逻辑电路又可以描述时序逻辑电路，还可以用元件调用方法描述逻辑功能 always 之间、assign 之间和实例引用，它们之间都是并行执行，always 内部是顺序执行。

assign 赋值语句中，被赋值的信号都是 wire 类型。assign 之所以被称为连续赋值，是因为它要不断检测表达式的变化。always 模块里被赋值的信号都要定义为 reg 类型，因为 always 可以反复执行，而 reg 表示信号的寄存，可以保留上次执行的值。

在赋值运算中，阻塞式赋值（=）表示在同一个 always 过程中，后面的赋值语句要等待前一个赋值语句执行完，后面的语句被该赋值语句阻塞。非阻塞式赋值（<=）表示在同一个 always 过程中，非阻塞赋值语句是同时进行的，排在后面的语句不会被该赋值语句所阻塞。块结束后才能完成赋值，块内所有非阻塞式赋值在 always 块结束时同时被赋值。在 always 过程中，begin…end 块内按先后顺序被立即赋值。

4．数据类型

Verilog HDL 的数据类型分为三类：线网类型、寄存器类型和参数类型。线网类型主要表示 Verilog HDL 中结构化元件之间的物理连线，其数值由驱动元件决定，如果没有驱动元件接到线网上，则其默认值为高阻状态 z；寄存器类型主要表示数据的存储单元，其默认值为不定值 x；参数类型常用参数来声明运行时的常数，可以用字符串表示的任何地方都可以用定义的参数来代替，参数是本地的，其定义只在本模块内有效。线网类型和寄存器类型最大的区别在于：寄存器类型数据保持最后一次的赋值，而线网类型数据则需要持续的驱动。

下面对本书常用到的数据类型进行介绍。wire 表示直通，即输入有变化，输出马上无条件地反映（如“与”、“非门”的简单连接）。reg 表示一定要有触发，输出才会反映输入的状态。reg 相当于存储单元，wire 相当于物理连线。reg 表示一定要有触发，没有输入的时候可以保持原来的值，但不直接与实际的硬件电路对应。对于 always 语句而言，赋值要申明成 reg；连续赋值 assign 的时候要用 wire。

5．关系运算符

==和!=这两个运算符只用于比较 0、1，遇到 z 状态或 x 状态时，结果都为 x 状态（x 在 if 中作为假条件）。

===和!==这两个运算符比较苛刻，属于 x 和 z 的精确比较，结果可能是 0、1。

其他关系运算符操作简单，这里就不再叙述。

6．逻辑运算符

&&和||是双目运算符，要求有两个操作数；而！是单目运算符，只要求一个操作数。

7．移位运算符

移位运算时，左移将保留高位，例如 4'b1000<<1 等于 5'b10000；右移则舍弃低位，例如 4'b0011>>1 等于 4'b0001。

8．跳转语句

if …else 有三种使用形式，如下所示。

使用形式 1：if (<条件表达式>)　语句或语句块；

使用形式 2：if (<条件表达式>)　语句或语句块 1；

　　else　语句或语句块 2；

使用形式 3：if (<条件表达式 1>) 语句或语句块 1；

　　else if (<条件表达式 2>) 语句或语句块 2；

　　……；

　　else if (<条件表达式 *n*>) 语句或语句块 *n*；

　　else　语句或语句块 *n*+1。

第三种形式适合描述优先编码器，if 条件中把状态 0/x/z 当成假，状态 1 当成真，非 0 的数值也当成真。

9．分支语句

case 语句是一个多路条件分支语句，case 语句有三种形式：case（四种状态的比较），casez（忽略 z 状态的比较），casex（忽略 x 和 z 状态的比较，只看哪些位的信号有用）。case 语句中所有表达式值的位宽必须相等。

10．循环语句

Verilog HDL 中有四种循环语句，包括 for 循环、while 循环、forever 循环、repeat 循环。其语法和用途和 C 语言相似。

11．语句块

有两种特殊的语句块：begin 顺序语句 ... end ，fork 并行语句... join，其差别在于块内语句的起止时间、执行顺序、相对延时等。块被命名后，因为变量都是静态的，其内部变量都可以被调用。

12．过程结构

initial 块只被无条件执行一次，always 块在满足条件时可以被不断执行；initial 块常用来写测试文件，always 块常用来进行电路描述，它既可以描述组合逻辑电路又可以描述时序逻辑电路，但如果后面有敏感信号列表则不能用 wait 语句。always 语句既可以描述电平触发又可以描述边沿触发，而 wait 语句只能描述电平触发；assign 常用于描述组合逻辑电路，测试文件中一般都是先用 initial 语句，后用 always 语句。

13．生成块

生成块的本质是使用循环内的一条语句代替多条重复的 Verilog HDL 语句，以便简化用户的编程。genvar 用于声明生成变量，生成变量只能用在生成块之间。

14．模块设计

一个模块设计包括 3 个部分：电路模块设计、测试模块设计和设计文档编写。设计者通过布局布线工具来生成具有布线延迟的电路，再进行仿真，得到时序分析报告。从时序分析

报告中可以知道电路的实际延时 t，同步电路内每个时钟周期要大于 t，从而可确定该运算逻辑的最高频率。

15．综合器

综合器之所以能够实现加法器和乘法器，是因为库中已经存在可配置的参数化器件模型，FPGA 内总线宽度可以自行定义以便实现高速数据流，三态数据总线相当于数据流的控制阀门。

16．同步时序逻辑和异步时序逻辑

同步时序逻辑指所有寄存器组由唯一时钟触发always@(posedge clk)或always@(negedage clk)；异步时序逻辑指触发条件不唯一，任意一个条件都会引起触发 always@(posedge clk or posedage reset)，目前的综合器是以同步时序逻辑进行综合的，它比异步时序逻辑更为可靠。

严格的同步要求时钟信号传递速度要远远大于各部分的延迟，实际中 clk 要单独用线，而不能经过反相器等部件。

表示同步时序电路的延时时，*#x* 表示延时 *x* 个时间单位，通常用于仿真测试。实际硬件延时时，长延迟采用计数器，小延迟采用 D 触发器，此方法用来取代延迟链。同步电路中，稳定的数据采集必须满足采样寄存器的建立和保持时间。延迟一般包括门延迟和线延迟。

17．状态机的开发步骤

首先，根据实际问题列出输入输出变量和状态数。其次，画出状态图并化简，写出状态转移真值表，得到逻辑表达式，用 D 触发器或 JK 触发器构建电路（目前多用 D 触发器）。最后，进行状态机的编码。

1.3.2　VHDL 语法要点

1．整体结构

```
entity 实体名 is
  port（端口说明）
end
architecture 结构体名 of 实体名 is
  说明部分;
begin
  代入语句;
  元件语句;
  进程语句;
end 结构体名;
```

2．端口说明

port（端口信号名：端口模式　数据类型名　[:=初始值]；…）;

3．端口模式

IN：从外部输入至实体，单向端口。OUT：从实体输出至外部，单向端口。INOUT：可

以从外部输入。BUFFER：可以从实体输出至外部，也可以从端口回读该输出值至实体，不可以从外部输入至实体。

4．结构体（architecture）

语法格式：

```
architecture 结构体名 of 设计实体名 is
    说明区；
begin
    执行语句区；
end  结构体名；
```

5．标识符

VHDL 语言的标识符可以是常数、变量、信号、端口、子程序或参数的名字。标识符可以用英文大/小写字母（首字符必须用字母）、数字“0～9”、下划线“_”表示。

6．数据类型

数据对象是 VHDL 语言中用于进行赋值等操作的客体。信号赋值可以采用 sig <= a + b 的形式来表示。主要包括如下数据类型。

强类型数据：如 Integer，Real，Bit 等已经在 VHDL 标准中预先定义，可直接使用，也可按照类型说明格式自定义用户所需的类型。

标量类型数据：如 Integer，Real，Bit，Boolean，Character，Time 等，在 STANDARD 程序包中定义。

复合类型数据：如数组Bit_vector 等，在 STANDARD 程序包中定义。

子类型数据：对一些已定义的数据类型进行一定的范围限制，从而形成一种特殊的数据类型。

7．运算符

算术运算符：如+（加）、–（减）、*（乘）、/（除）、MOD（取模）、REM（取余）、**（乘方）、ABS（取绝对值）。

关系运算符：如=（相等）、/=（不等）、<（小于）、<=（小于等于）、>（大于）、>=（大于等于）。

逻辑运算符：如 NOT（非）、AND（与）、NAND（与非）、OR（或）、NOR（或非）、XOR（异或）。

符号运算符：如+（正号）、–（负号）。

连接运算符：如&（可用于连接字符串或位串）。

8．表达式

算术表达式：如 a + b。

关系表达式：如 x < y。

逻辑表达式：如 a1 AND a 2。

9．并行语句

VHDL 语言实现的许多操作都是并行执行的，并行语句用来描述并行发生的行为。结构体由一至多个并行语句构成，这些语句之间是并行执行的。基本的并行语句可分为四种形式：直接设置语句、条件式信号设置语句、选择式信号设置语句和进程语句。

（1）直接设置语句

直接设置语句是采用“<=”运算符。例如：

D<= not A；

E<=B and C；

F<=A or B or C；

这三条语句虽然是分三行写的，但实际上三条语句是同时执行的。

（2）条件式信号设置语句：When-Else

When-Else 命令也是属于并行的语句命令，它的语法格式如下。

信号 A <= 信号 B When（条件 1）Else
　　信号 C When（条件 2）Else
　　信号 D；

上述的条件式，是指一般常见的布尔表达式，亦即条件式的结果必定是真（True）或错（False）中的一种：语法中的条件式 1 为 True 时，则将信号 B 传递给信号 A；否则再确认条件式 2 为 True 时，将信号 C 传递给信号 A；如果条件 1 和条件 2 都不成立，将信号 D 的值传递给信号 A。

（3）选择式信号设置语句：With-Select

语法格式如下。

With 选择信号 X Select
信号 Y <= 信号值 A When 选择信号 X 值为 m，
　　信号值 B When 选择信号 X 值为 n，
　　…
　　信号值 Z When Others；

With-Select 的命令作用是，判断选择信号 X 的值，依次是 m 或 n 的相应条件值，然后再将它对应的信号值 A 或信号值 B 传递给信号 Y；上述 With-Select 语法命令中的 m，n 等值要求互不相同。

（4）进程（Process）语句

Process（进程）语句是一种并行处理语句，在一个构造体中多个 Process 语句可以同时并行运行。因此，Process 语句是 VHD 语言中描述硬件系统并行行为最基本的语句。其语法结构如下。

```
[进程名]：Process （信号 1，信号 2，…）
Begin
    …
End Process;
```

进程 Process 语句中总是带有 1 个或几个信号量，这些信号是 PROCESS 的输入信号（不

一定是所有的输入信号），在 VHDL 语言中也称为敏感量。这些信号无论哪一个发生变化（如由“0”变“1”或由“1”变“0”）都将启动该 Process 语句。一旦启动以后，Process 中的语句将从上到下逐句执行一遍，当最后一条语句执行完毕以后，就返回到开始的 Process 语句，等待下一次变化的出现。当没有敏感量时，进程将无限循环执行。

10．顺序语句

所谓顺序语句就是语句是按先后顺序执行的。顺序描述语句只能出现在并行语句 Process 中，顺序语句包括以下语句。

赋值语句：信号赋值语句和变量赋值语句。

分支控制语句：条件 if 语句块和选择 case 语句块。

循环控制语句：主要有 while 条件表达式 loop；for 循环步进条件 loop；Next 语句；Exit 语句。

同步控制语句：主要有 wait　on；wait　until；wait　for。

1.4 FPGA 开发环境简介

FPGA 集成开发环境 ISE（Integrated Software Environment，集成软件环境）包括设计输入、仿真、综合、布局布线、生成 BIT 文件、配置、在线调试等功能，支持 ModelSim、Synplify 等多种第三方工具，涵盖了 FPGA 开发的全过程。ModelSim 是唯一的单内核支持 VHDL 和 Verilog 混合仿真的仿真器。ISE 中可直接调用 ModelSim 仿真，也可以独立使用 ModelSim 进行仿真。

1.4.1 FPGA 开发环境 ISE

Xilinx 公司的 ISE 系列是不断更新升级的，下面以 ISE10 系列为例，介绍此软件。ISE10 主界面如图 1.2 所示。

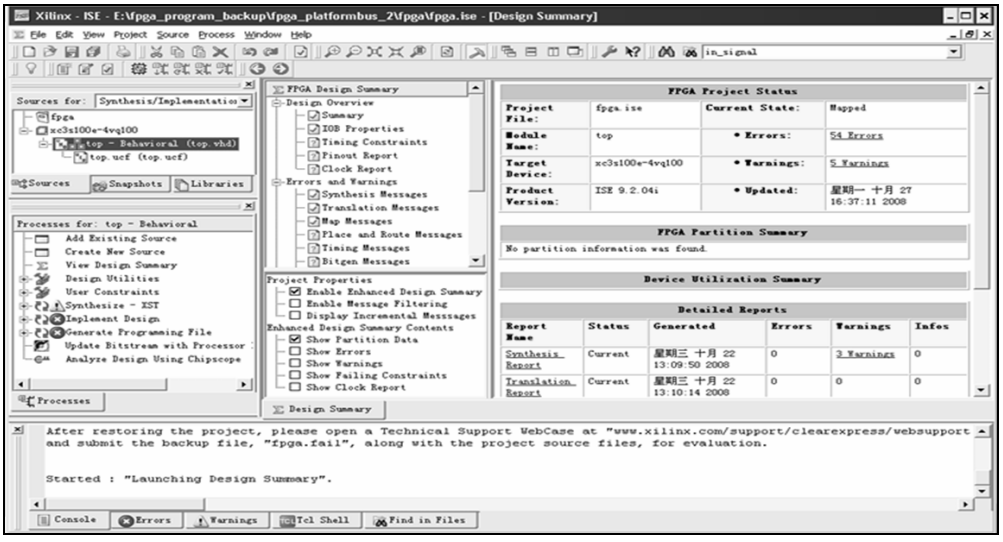

图 1.2　ISE10 主界面

ISE 的主要功能包括设计输入、综合、仿真、实现和下载，涵盖了可编程逻辑器件开发的全过程。从功能上讲，完成 CPLD/FPGA 的设计流程无需借助任何第三方电子设计自动化（Electronic Design Automation，EDA）软件。下面简要说明各功能的作用。

① 设计输入。ISE 提供的设计输入工具包括用于 HDL 代码输入和查看报告的 ISE 文本编辑器（ISE Text Editor），用于原理图编辑的工具，用于生成 IP Core 的 Core Generator，用于状态机设计的 StateCAD 以及用于约束文件编辑的 Constraint Editor 等。

② 综合。ISE 的综合工具不但包含了 Xilinx 自身提供的综合工具 XST，同时还可以内嵌 Mentor Graphics 公司的 Leonardo Spectrum 和 Synplicity 公司的 Synplify，实现无缝链接。

③ 仿真。ISE 本身自带了一个具有图形化波形编辑功能的仿真工具 HDL Bencher，同时又提供了使用 Model Tech 公司的 ModelSim 进行仿真的接口。

④ 实现。此功能包括了翻译、映射、布局布线等，还具备时序分析、管脚指定以及增量设计等高级功能。

⑤ 下载。下载功能包括了 BitGen，用于将布局布线后的设计文件转换为位流文件，还包括了 IMPACT，功能是进行芯片配置和通信，控制将程序烧写到 FPGA 芯片中去。

ISE 主界面窗口功能概述如下。

① 左上角的窗口是源文件窗口，设计工程所包括的文件以分层的形式列出。

这个窗口有三个标签：源（Source）、快照（Snapshots）、库（Library）。源标签内显示工程名、指定的芯片和设计相关文档。设计视图的每一个文件都有一个相关的图标，这个图标显示的是文件的类型（HDL 文件、原理图、IP 核和文本文件）。“+”表示该设计文件包含了更低层次的设计模块，标签内显示的是当前所打开文件快照，一个快照是在该工程里所有文件的一个拷贝。通过该标签可以查看报告、用户文档和源文件，该标签下所有的信息为只读，库标签内显示与当前工程相关的库。

② 在该子窗口的下面是处理窗口，该窗口描述的是对于选定的设计文件可以使用的处理流程。

该窗口只有一个处理标签，该标签有下列功能：添加已有文件，创建新文件，查看设计总结（访问符号产生工具，例化模板，查看命令行历史和仿真库编辑），用户约束文件（访问、编辑位置和时序约束），综合（检查语法、形成综合报告），设计实现（访问实现工具，设计流程报告和其他一些工具），产生可编程文件（访问配置工具和产生比特流文件）。

③ ISE 主界面最下面是脚本窗口，在该窗口中显示了消息、错误和警告的状态，同时还有 Tcl 脚本的交互和文件中查找的功能。

脚本子窗口有 5 个默认标签：Console，Error，Warnings，Tcl shell，Find in file。Console 标签显示错误、警告和信息，“X”表示错误，“!”表示警告；Error 标签只显示错误消息；Warning 标签只显示警告消息；Tcl shell 标签是设计人员的交互控制台，除了显示错误、警告和信息外，还允许输入 ISE 特定命令；Find in file 标签显示的是选择 Edit→Find in File 操作后的查询结果。

④ ISE 的右上角是多文档的窗口，在该窗口可以查看 html 的报告，ASCII 码文件、原理图和仿真波形。通过选择 View→Restore Default Layout 可以恢复界面的原始设置。

另外，工作区子窗口提供了设计总结、文本编辑器、ISE 仿真器/波形编辑器、原理图编辑器功能。设计总结提供了关于该设计工程的更高级信息，包括信息概况、芯片资源利用报

告、与布局布线相关的性能数据、约束信息和总结信息等。源文件和其他文本文件可以通过设计人员指定的编辑工具打开。编辑工具的选择由 Edit→Preference 属性决定，默认是 ISE 文本编辑器，通过该编辑器可以编辑源文件和用户文档，也可以访问语言模板。

ISE 的开发流程描述如下。

① 新工程建立如图 1.3 所示。

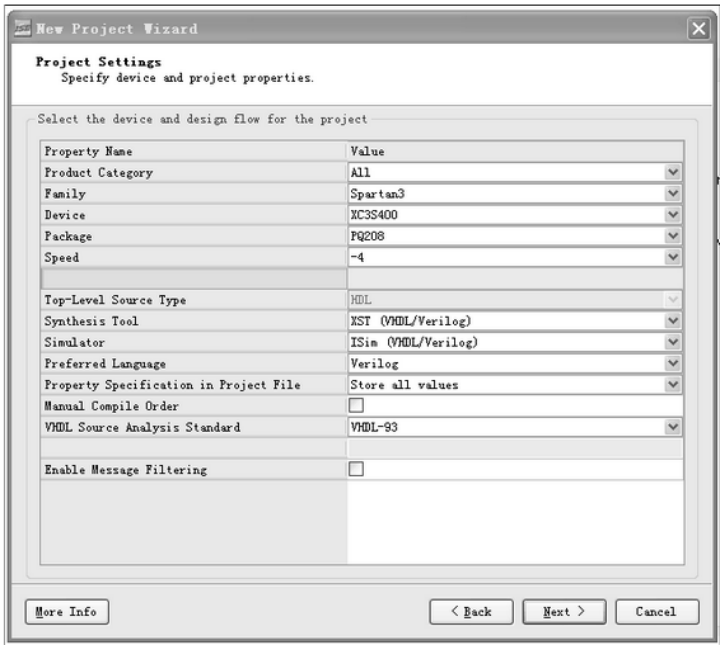

图 1.3 新工程建立

② 选择源码类型如图 1.4 所示。

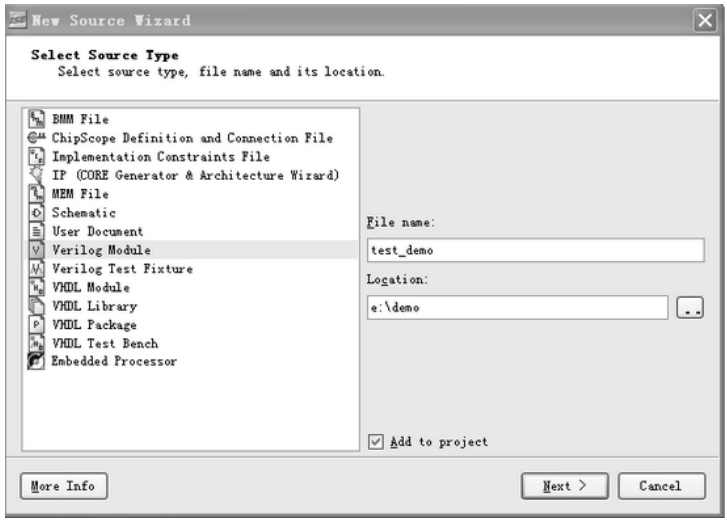

图 1.4 选择源码类型

③ 输入代码，建立测试文件如图 1.5 所示。

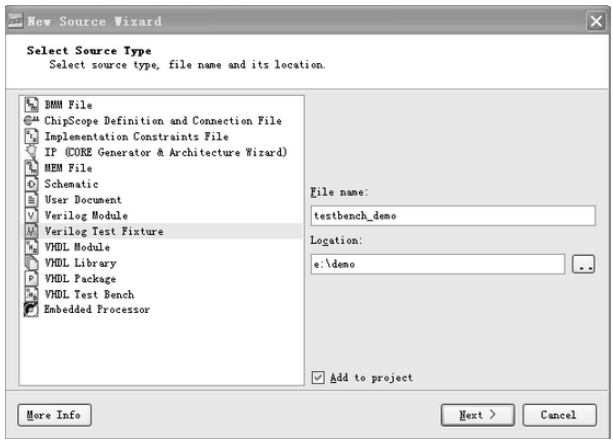

图 1.5　建立测试文件

④ 仿真结果如图 1.6 所示。

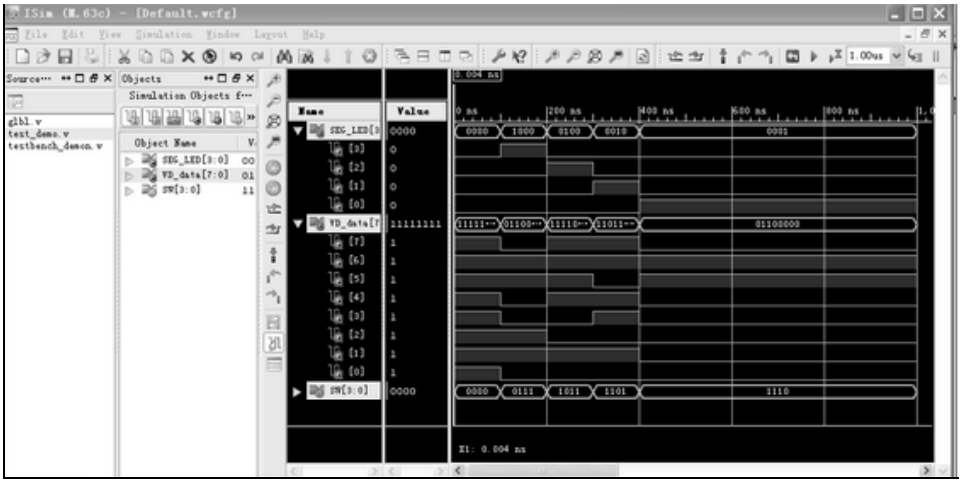

图 1.6　仿真结果

⑤ 建立约束文件如图 1.7 所示。

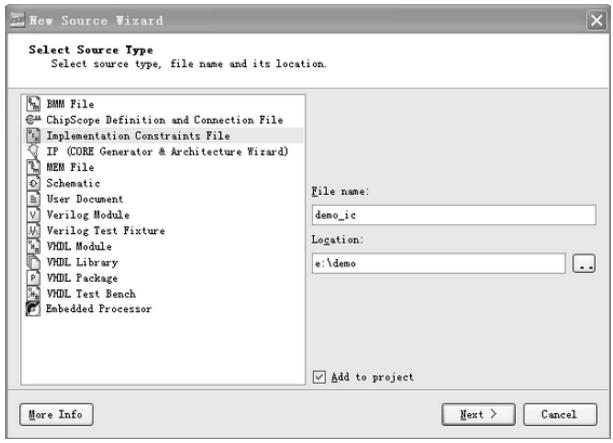

图 1.7　建立约束文件

⑥ 综合如图 1.8 所示。

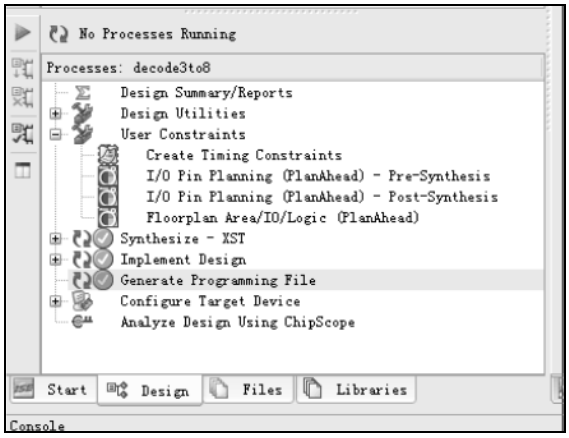

图 1.8 综合

1.4.2 FPGA 开发环境 ModelSim

ModelSim 的功能仿真只验证其功能是否正确，在时序上不做验证。在进行功能仿真时还需要注意，信号通过某个网络时是存在延迟的，而在功能仿真时不会体现出来，输入信号的改变会立即在输出端反映出来。所以要明确功能仿真和时序仿真是有区别的。

要对模块进行仿真，首先要建立仿真测试激励文件，ModelSim 根据测试激励对设计进行仿真。而 ISE 提供了一个相当实用的工具——HDL Bencher（HDL 测试台），HDL Bencher 可以方便地定义想要的输入波形和输出波形，仿真后 ModelSim 还可以将实际仿真的结果与测试者提供的波形进行比较，并且提示错误信息。下面简单介绍一下 ModelSim 的功能仿真过程。

① 新建测试激励文件如图 1.9 所示。

图 1.9 新建测试激励文件

② 选择与测试激励文件相关联的源文件如图 1.10 所示。

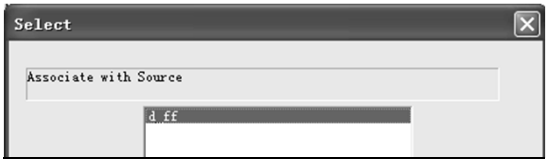

图 1.10　选择与测试激励文件相关联的源文件

③ 时钟选择对话框如图 1.11 所示。

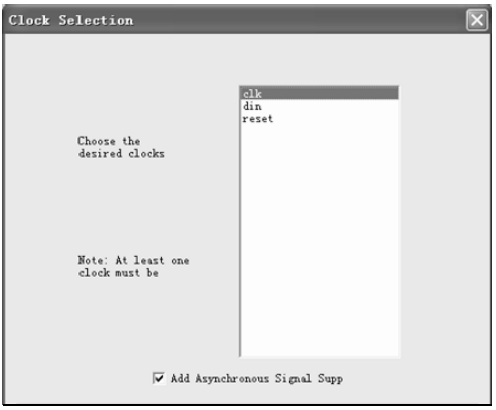

图 1.11　时钟选择对话框

④ 时钟设置如图 1.12 所示。

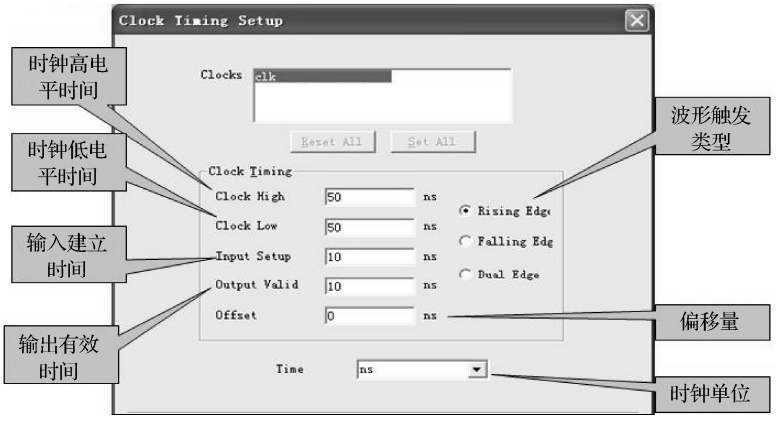

图 1.12　时钟设置

⑤ 仿真后的波形如图 1.13 所示。

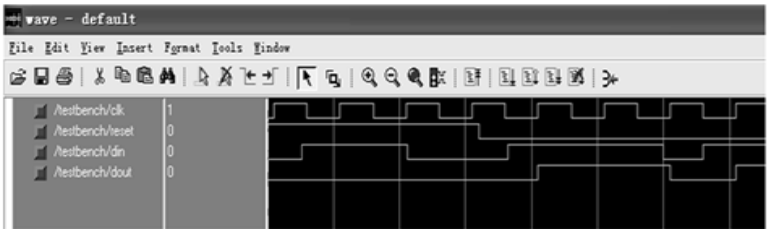

图 1.13　仿真后的波形

1.5 密码算法的 FPGA 实现流程

密码算法 FPGA 的实现方法和实现流程，需要通过实际的设计过程加深理解。

1.5.1 FPGA 一般实现流程

FPGA 开发流程如图 1.14 所示。

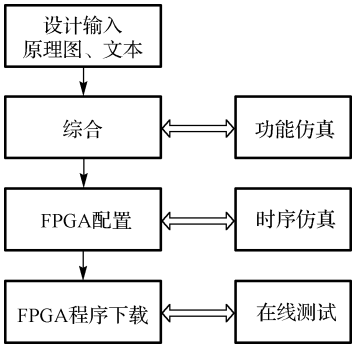

图 1.14 FPGA 开发一般流程

设计时应注意如下几方面内容。

（1）设计尽量文档化。要将设计思路，详细实现等写入文档，然后经过严格讨论和评审通过，才能进行下一步的工作。

（2）端口信号排列要统一。一个信号只占一行，最好按“从哪个模块来到哪个模块去”的关系排列。内部模块不能出现 inout 端口，如果需要，把 inout 端口拆分为一组 input 和 output。

（3）信号的命名要清晰明了。有明确含义，同时使用完整的单词或大家基本可以理解的缩写，避免使人产生误解。

（4）一个模块尽量只用一个时钟，这里的一个模块是指一个 module。在多时钟域的设计中如果涉及跨时钟的情况，最好有专门一个模块做时钟隔离，这样做可以让综合器综合出更优的结果。除了低功耗设计，一般不要用门控时钟，因为这会增加设计的不稳定性。尽量不要用计数器分频后的信号做其他模块的时钟，否则时钟太多会对设计的可靠性极为不利，也大大增加了静态时序分析的复杂性。

（5）尽量在底层模块上做逻辑，在高层上尽量做例化，不要出现任何胶连逻辑（glue logic），哪怕仅仅对某个信号取反。数据都要用十六进制或二进制表示，且要标明位数，这样综合器综合出的结果较好。

（6）一般来说，进入 FPGA 的信号必须先同步。所有模块的输出都要寄存器化，以提高工作频率，这对时序收敛也是有好处的。

1.5.2 密码算法的 FPGA 实现流程

密码算法的设计实现中，并不是每一个算法都适合用 FPGA 实现。用 FPGA 实现密码算法的优点是，密码算法可以并行计算，因此，在算法设计时，多用并行计算，少用串行计算。

同时，根据速度要求，要注意算法的深度。比如，DES 算法便于 FPGA 实现，不便于软件操作，而 AES 算法两者皆可。

对 FPGA 选择时，要根据密码算法的规模来选择，如果选择的规模太大，会造成不必要的浪费；规模太小，就不能实现密码算法；还要考虑性价比。所以，要综合这些因素来考量 FPGA 的选择。

FPGA 实现要进行优化选择，对同一基本运算，用 FPGA 实现有多种方法，必须根据实际情况使用最优化的方法。比如运算速度、占用空间大小等，对整个结构要进行综合考虑。

密码算法的输出：在用 FPGA 实现算法输出时，要考虑 FPGA 实现的算法是专用还是通用，如专用要考虑使用专用接口；如通用则要考虑各种使用情况，这时，就要预留一定的缓存空间。

其流程实现如图 1.15 所示。

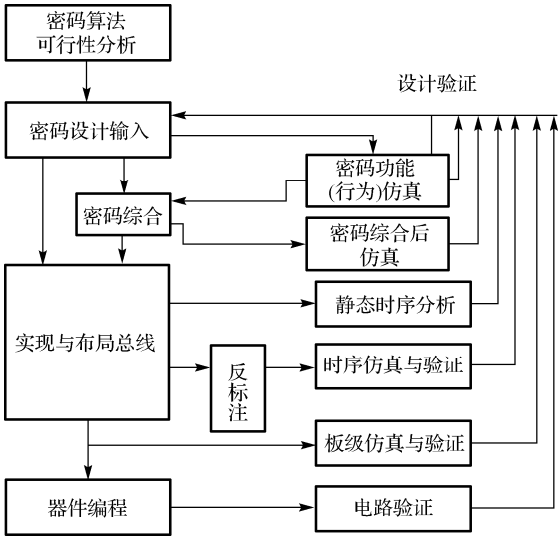

图 1.15　密码算法的 FPGA 实现流程

1.6 本章小结

本章首先讲述了 Xilinx 公司 FPGA 的发展概述，Xilinx 公司不断推陈出新，一直保持着 FPGA 领域的领先地位；其次讲解了 FPGA 的工作原理，然后简单叙述了书中用到的 Verilog HDL 和 VHDL 语言的相关语法知识；再次，介绍了 ISE 和 ModelSim 的相关原理；最后，总结出密码算法的 FPGA 实现流程，提出 FPGA 系列器件的选取要根据工程规模和占用空间等因素综合考量。

第 2 章　DES 算法 FPGA 实现

数据加密标准（Data Encryption Standard，DES）算法是 1971 年美国学者塔奇曼（Tuchman）和迈耶（Meyer）根据信息论创始人香农（Shannon）提出的“多重加密有效性理论”而创立的。1977 年 1 月 15 日，美国正式公布 DES 为数据加密标准算法。DES 是一个迭代的分组密码，使用 Feistel 网络结构。由于 DES 算法的密钥短，攻破相对容易，所以已被 AES 算法（将在第 3 章介绍）替代。

2.1　DES 算法原理

2.1.1　参数产生

DES 使用一个 56 位的密钥以及附加的 8 位奇偶校验位，产生最大 64 位的分组大小。DES 算法输入是 64 位的明文，在 64 位密钥的控制下产生 64 位的密文。64 位明文组在加密之前，要先经过一个初始置换，初始置换记为 IP。密文组在解密之前也要先进行逆初始置换，逆初始置换记为 IP^{-1}。不难知道，IP^{-1} 是 IP 的逆。初始置换 IP 用于对明文 m 中的各位进行换位，目的在于打乱明文 m 中各位的次序。经过初始置换后，m 变为

$$m' = m'_1 m'_2 \cdots m'_{64} = m_{58} m_{50} \cdots m_7$$

即 m 中的第 58 位变为 m' 中的第 1 位，m 中的第 50 位变为 m' 中的第 2 位，以此类推，最后将 m 中的第 7 位变为 m' 中的第 64 位。

2.1.2　密钥生成

设密钥 $k = k_1 k_2 \cdots k_{64}, k_i \in \{0,1\}$，$1 \leqslant i \leqslant 64$。密钥 k 中有 8 位是奇偶校验位，分别位于第 8，16，24，32，40，48，56，64 位。奇偶校验位用于检查密钥 k 在产生和分配以及存储过程中可能发生的错误。选择置换 PC-1 用于去掉密钥 $\mathrm{rk}_i (i = 0,1,2,3)$ 中的 8 个奇偶校验位，并对其余的 56 位打乱重新排列。将 PC-1 输出中的前 28 位作为 C_0，后 28 位作为 D_0。C_0 中的各位从左到右依次为密钥 $m > 2b_n \geqslant \sum_i b_i$ 中的第 57，49，…，44，36 位，D_0 中的各位从左到右依次为密钥 k 中的第 63，55，…，12，4 位。对于 $1 \leqslant i \leqslant 16$，

$$C_i = \mathrm{LS}_i(C_{i-1})$$
$$D_i = \mathrm{LS}_i(D_{i-1})$$

其中，LS_i 表示对 C_{i-1} 和 D_{i-1} 进行循环左移变换，$\mathrm{LS}_1, \mathrm{LS}_2, \mathrm{LS}_9, \mathrm{LS}_{16}$ 是循环左移 1 位变换，其余的 LS_i 是循环左移 2 位变换。选择置换 PC-2 用于从 $C_i D_i$ 中选取 48 位作为子密钥 K_i，$C_i D_i$ 表

示从左到右将 D_i 排在 C_i 的后面，C_iD_i 的长度为 56 位密钥，子密钥 K_i 中的各位从左到右依次为 C_iD_i 中的第 14,17, …, 29,32 位。

2.1.3　加密解密过程

首先对 64 位输入 X 进行一次初始置换 IP，以打乱原来的次序。对初始置换 IP 后的数据分成左右两半，左边记为 L_0，右边记为 R_0，对 R_0 实行在子密钥控制下的 f 变换，其结果记为 $f(R_0, K_1)$，得到的 32 比特输出再与 L_0 做逐位异或（XOR）运算，其结果成为下一轮的 R_1，R_0 则成为下一轮的 L_1。对 L_1、R_1 实行的运算过程与 L_0、R_0 相同，结果为 L_2、R_2，如此循环 16 次，最后得 L_{16}、R_{16}。运算过程可用公式简洁地表示如下：

$$\begin{cases} R_i = L_{i-1} \oplus f(R_{i-1}, K_i) \\ L_i = R_{i-1} \qquad (i = 1,2,\cdots,16) \end{cases}$$

其中 $\oplus$ 是按位做不进位加法运算（异或），再对 64 位数字 L_{16}、R_{16} 做初始置换的逆置换 IP^{-1}，即得密文 Y，如图 2.1 所示。

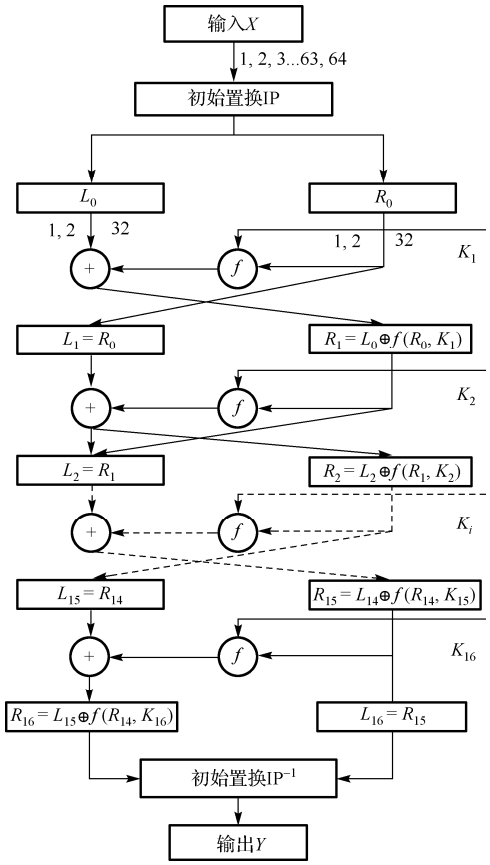

图 2.1　DES 算法的加、解密过程

在 16 次加密后并未交换 L_{16} 和 R_{16}，而是直接将 R_{16} 和 L_{16} 作为逆初始置换 IP^{-1} 的输入，这样做使得 DES 的解密和加密完全相同，在以上过程中只需输入密文并反序输入子密钥，最后获得的就是相应的明文，这就是解密的过程。

其中 f 函数是整个 DES 加密算法中最重要的部分，而其中的重点在 S 盒上。

S 盒是 DES 算法中唯一的非线性部件，当然也是整个算法的安全性所在。每个 S 盒是 6 位输入、4 位输出的变换，其变换规则为：取{0，1，…，15}上的 4 个置换，即它的 4 个置换并列排成 4 行，得到一个 4×16 的矩阵。若给定该 S 盒的输入 $b_0b_1b_2b_3b_4b_5$，其输出对应该矩阵第 L 行 n 列对应数的二进制表示，其中 L 为由 b_0b_5 形成的二进制表示，n 为由 $b_1b_2b_3b_4$ 形成的二进制表示，这样，每个 S 盒可用一个 4×16 矩阵或数表来表示。

2.1.4 安全性分析

密钥安全问题。个别密钥不适合作为加密密钥，比如子密钥生成器里面全为 0 或者 1 的密钥，不管如何循环位移编码总是不变的，以至于 16 轮加密步骤所使用的子密钥是恒定不变的，使得安全性失去保障。

DES 算法密钥过短。DES 算法经受住了时间的考验，但是在差分分析法或线性逼近法的理论下，联合多台工作站同时协同工作，不到半个月就可以找到密钥。DES 受制于此的原因在于密钥太短，只有 56 位，即 7 个英文字符，这成为 DES 被遍历搜索破译的短板。

在 DES 中，初始置换 IP 和逆初始置换 IP^{-1} 各使用一次，使用这两个置换的目的是把数据彻底打乱重新排列。它们对数据加密所起作用不大，因为它们与密钥无关，置换关系固定，所以一旦公开，它们对数据的加密便无多大价值。

在 DES 中，除了 S 盒是非线性变换外，其余变换均为线性变换，因此，S 盒是 DES 的关键。可以看出，任意改变 S 盒输入中的几位，其输出至少有两位发生变化。由于在 DES 中使用了 16 次迭代，所以即使改变明文或密钥中的 1 位，密文中也有大约 32 位发生变化。

在 DES 中，子密钥的产生也很有特色，它确保密钥中各位的使用次数基本相等。实验表明，56 位密钥中每位的使用次数在 12～15 次之间。

随着密码分析技术和计算能力的提高，DES 的安全性受到质疑和威胁。密钥较短是 DES 的一个主要缺陷，DES 不能抵御对密钥的穷举搜索攻击。目前，美国已经正式公布实施高级加密标准 AES 用于取代 DES。

2.2 DES 算法相关模块的 FPGA 设计

本节中 DES 算法的 FPGA 设计，使用硬件描述语言 Verilog HDL 来实现，所设计的模块主要有 IP 和 IP^{-1} 模块、密钥扩展模块、S 盒模块、f 函数模块和 DES 算法的顶层模块，结构图如图 2.2 所示。

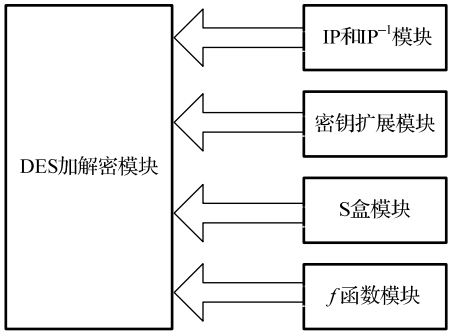

图 2.2 DES 算法的关键模块

2.2.1　IP 和 IP^{-1} 模块设计

DES 在对明文进行初始置换（IP）之后，执行 16 轮的迭代操作，最后经初始置换的逆变化（IP^{-1}），从而得到密文。所谓迭代操作是在密钥控制下，多次利用轮函数 f 进行加密变换，以实现扩散和混淆的效果。IP 和 IP^{-1} 模块的查找表设计如程序 2-1 所示。

程序 2-1：

```
IP= {IP [58],IP [50],IP [42],IP [34],IP [26],IP [18],IP [10],IP [2],
     IP [60],IP[52],IP[44],IP[36],IP[28],IP[20],IP[12],IP[4],
     IP[62],IP[54],IP[46],IP[38],IP[30],IP[22],IP[14],IP[6],
     IP[64],IP[56],IP[48],IP[40],IP[32],IP[24],IP[16],IP[8],
     IP[57],IP[49],IP[41],IP[33],IP[25],IP[17],IP[9],IP[1],
     IP[59],IP[51],IP[43],IP[35],IP[27],IP[19],IP[11],IP[3],
     IP[61],IP[53],IP[45],IP[37],IP[29],IP[21],IP[13],IP[5],
     IP[63],IP[55],IP[47],IP[39],IP[31],IP[23],IP[15],IP[7]};
IVIP = {IP_1[40],IP_1[8],IP_1[48],IP_1[16],IP_1[56],IP_1[24],IP_1[64],IP_1[32],
      IP_1[39],IP_1[7],IP_1[47],IP_1[15],IP_1[55],IP_1[23],IP_1[63],IP_1[31],
      IP_1[38],IP_1[6],IP_1[46],IP_1[14],IP_1[54],IP_1[22],IP_1[62],IP_1[30],
      IP_1[37],IP_1[5],IP_1[45],IP_1[13],IP_1[53],IP_1[21],IP_1[61],IP_1[29],
      IP_1[36],IP_1[4],IP_1[44],IP_1[12],IP_1[52],IP_1[20],IP_1[60],IP_1[28],
      IP_1[35],IP_1[3],IP_1[43],IP_1[11],IP_1[51],IP_1[19],IP_1[59],IP_1[27],
      IP_1[34],IP_1[2],IP_1[42],IP_1[10],IP_1[50],IP_1[18],IP_1[58],IP_1[26],
      IP_1[33],IP_1[1],IP_1[41],IP_1[9],IP_1[49],IP_1[17],IP_1[57],IP_1[25]};
```

IP 的逆变换在程序中用 IVIP 表示，IP 和 IP^{-1} 模块运用查找表的方式编写。

2.2.2　密钥扩展设计

DES 算法中初始 Key 值为 64 位，但密钥实际可用位数只有 56 位，其余 8 位为奇偶校验位。PC-1 变换和 PC-2 变换以及循环左移操作都是位替换操作。设计思路如程序 2-2 所示。

程序 2-2：

```
assign K = {    key[01:07], key[09:15], key[17:23], key[25:31],
                key[33:39], key[41:47], key[49:55], key[57:63]      };
assign K1[1] = K[47];                assign K1[2] = K[11];
assign K1[3] = K[26];                assign K1[4] = K[3];
assign K1[5] = K[13];                assign K1[6] = K[41];
assign K1[7] = K[27];                assign K1[8] = K[6];
assign K1[9] = K[54];                assign K1[10] = K[48];
assign K1[11] = K[39];               assign K1[12] = K[19];
assign K1[13] = K[53];               assign K1[14] = K[25];
assign K1[15] = K[33];               assign K1[16] = K[34];
assign K1[17] = K[17];               assign K1[18] = K[5];
assign K1[19] = K[4];                assign K1[20] = K[55];
assign K1[21] = K[24];               assign K1[22] = K[32];
```

```
assign K1[23] = K[40];              assign K1[24] = K[20];
assign K1[25] = K[36];              assign K1[26] = K[31];
assign K1[27] = K[21];              assign K1[28] = K[8];
assign K1[29] = K[23];              assign K1[30] = K[52];
assign K1[31] = K[14];              assign K1[32] = K[29];
assign K1[33] = K[51];              assign K1[34] = K[9];
assign K1[35] = K[35];              assign K1[36] = K[30];
assign K1[37] = K[2];               assign K1[38] = K[37];
assign K1[39] = K[22];              assign K1[40] = K[0];
assign K1[41] = K[42];              assign K1[42] = K[38];
assign K1[43] = K[16];              assign K1[44] = K[43];
assign K1[45] = K[44];              assign K1[46] = K[1];
assign K1[47] = K[7];               assign K1[48] = K[28];
```

密钥为 64 位初始密钥变量，*K* 为去掉了 8 个奇偶校验位后的初始密钥变量。在 Verilog HDL 中，assign 相当于连线，它的用处是将一个变量的值不间断地赋给另外一个变量，密钥扩展使用 assign 语句来实现。

2.2.3 S 盒设计

S 盒是以表格的形式给出的，因此可以设计成查找表。在 Verilog HDL 中使用 case 语句可以实现查找表设计。将 S 盒输入变量作为 case 语句的输入，根据每个 S 盒中的内容，列出所有取值情况即可。S 盒设计如程序 2-3 所示。

程序 2-3：

```
always @(addr) begin
   case ({addr[6],addr[1], addr[5:2]})
0:  dout =  14;      1:  dout =   4;      2:  dout =  13;
3:  dout =   1;      4:  dout =   2;      5:  dout =  15;
6:  dout =  11;      7:  dout =   8;      8:  dout =   3;
9:  dout =  10;      10:  dout =   6;     11:  dout =  12;
12:  dout =   5;     13:  dout =   9;     14:  dout =   0;
15:  dout =   7;     16:  dout =   0;     17:  dout =  15;
18:  dout =   7;     19:  dout =   4;     20:  dout =  14;
21:  dout =   2;     22:  dout =  13;     23:  dout =   1;
24:  dout =  10;     25:  dout =   6;     26:  dout =  12;
27:  dout =  11;     28:  dout =   9;     29:  dout =   5;
30:  dout =   3;     31:  dout =   8;     32:  dout =   4;
33:  dout =   1;     34:  dout =  14;     35:  dout =   8;
36:  dout =  13;     37:  dout =   6;     38:  dout =   2;
39:  dout =  11;     40:  dout =  15;     41:  dout =  12;
42:  dout =   9;     43:  dout =   7;     44:  dout =   3;
45:  dout =  10;     46:  dout =   5;     47:  dout =   0;
48:  dout =  15;     49:  dout =  12;     50:  dout =   8;
51:  dout =   2;     52:  dout =   4;     53:  dout =   9;
```

```
        54:  dout =   1;      55:  dout =   7;      56:  dout =   5;
        57:  dout =  11;      58:  dout =   3;      59:  dout =  14;
        60:  dout =  10;      61:  dout =   0;      62:  dout =   6;
        63:  dout =  13;
        endcase
        end
        endmodule
```

例如程序中输入的{addr[6],addr[1], addr[5:2]}，二进制表示为 001011，表示为十进制时为 11，于是输出为 12，查找上表可知对应输出为 12，与结果一致。S2～S8 盒的结构与 S1 盒设计方式相同，只是 case 语句中各分支的输出变量取值不同。

2.2.4　*f* 函数设计

f 函数是整个 DES 加密中最重要的部分。*f* 函数对 32 位数据和 48 位的子密钥进行变换运算。首先将 32bit 数据 R_i(i=1，2，…，16)进行 E 置换扩展为 48bit 数据，第 32 位换到第 1 位，第 1 位换到第 2 位，以此类推，得到 48 位的置换结果。在 Verilog HDL 中，置换过程可以用 assign 语句和位拼接运算符“{}”实现。关键的实现过程框架如程序 2-4 所示。

程序 2-4：

```
    assign E = { INR[32], INR[1], INR[2], INR[3], INR[4], INR[5], INR[4], INR[5],
                 INR[6], INR[7], INR[8], INR[9], INR[8], INR[9], INR[10], INR[11],
                 INR[12], INR[13], INR[12], INR[13], INR[14], INR[15], INR[16],
                 INR[17], INR[16], INR[17], INR[18], INR[19], INR[20], INR[21],
                 INR[20], INR[21], INR[22], INR[23], INR[24], INR[25], INR[24],
                 INR[25], INR[26], INR[27], INR[28], INR[29], INR[28], INR[29],
                 INR[30], INR[31], INR[32], INR[1]};
    assign X = E ^ K_sub;
                 sbox1 u0( .addr(X[01:06]), .dout(S[01:04]) );
                 sbox2 u1( .addr(X[07:12]), .dout(S[05:08]) );
                 sbox3 u2( .addr(X[13:18]), .dout(S[09:12]) );
                 sbox4 u3( .addr(X[19:24]), .dout(S[13:16]) );
                 sbox5 u4( .addr(X[25:30]), .dout(S[17:20]) );
                 sbox6 u5( .addr(X[31:36]), .dout(S[21:24]) );
                 sbox7 u6( .addr(X[37:42]), .dout(S[25:28]) );
                 sbox8 u7( .addr(X[43:48]), .dout(S[29:32]) );
    assign OUTP = { S[16], S[7], S[20], S[21], S[29], S[12], S[28],
                    S[17], S[1], S[15], S[23], S[26], S[5], S[18],
                    S[31], S[10], S[2], S[8], S[24], S[14], S[32],
                    S[27], S[3], S[9], S[19], S[13], S[30], S[6],
                    S[22], S[11], S[4], S[25]}^ INL;
    assign OUTL = INR;
```

程序代码中的 sbox1 到 sbox8 是调用 8 个 S 盒的过程，结果分为 8 个部分，将分别对应 8 个 S 盒的输出。

2.2.5 顶层模块设计

S 盒模块、密钥扩展模块和 f 函数模块设计完成后，就可以去实现该算法的顶层模块了。为了实现 DES 算法的 16 轮迭代，顶层调用模块一般需要设计成时序逻辑电路，利用状态机对 f 函数进行调用，可以很好地控制算法的执行。因为以上各模块设计中没有初始置换 IP、逆初始值换 IP^{-1} 以及每一轮的数据交换过程，所以在顶层模块设计时需要考虑这些内容。算法工作过程为：算法启动，开始依次读入数据和初始密钥值，再根据加/解密标志进行加/解密操作；算法执行结束后，输出加/解密结果，同时给出算法结束标志。程序框架如下所示，其中 IP()和 IVIP()是定义置换 IP 和置换 IP^{-1} 的函数。

程序 2-5：

```
always @(posedge clk)
begin
if(reset == 0)
 begin
    ...
   end
else
    case(state)
         s0 : begin din1  = datain; state = s1; end
         s1 : begin din2  = datain; state = s2; end
         s2 : begin key1  = datain; state = s3; end
         s3 : begin key2  = datain; state = s4; end
         s4 : begin
                 {din1, din2} = IP({din1, din2});
                 if(sign == 0) begin roundsel = 0; state = s5; end
                 else         begin roundsel = 15;state = s7; end
              end
         s5 : begin //encription
                 {din1, din2} = {cipherout1, cipherout2};
                 roundsel = roundsel + 1;
                 state = s6;
              end
         s6 : begin
                 if(roundsel == 15)
                     begin
                         {din1, din2} = IVIP({cipherout2, cipherout1});
                         ready = 1;
                         state = s10;
                     end
                 else state = s5;
              end
         s7 : begin//decription
```

```
                    {din1, din2} = {cipherout1, cipherout2};
                    roundsel = roundsel - 1;
                    state = s8;
                end
            s8 : begin
                    if(roundsel == 0)
                        begin
                            {din1, din2} = IVIP({cipherout2, cipherout1});
                            ready = 1;
                            state = s10;
                        end
                    else state = s7;
                end
                    s10: begin dataout = din1; state = s11; end
            s11: begin dataout = din2; state = s12; end
            s12: begin ready = 1; state = s12; end
            default : state = s12;endcase
    end
    crp c1(.OUTR(cipherout2),.OUTL(cipherout1),.INR(din2),.INL(din1),.K_s ub(key));
    key_sel c2(.K_sub(key), .key({key1, key2}), .roundSel(roundsel));
```

程序中主要接口信号定义如下。

reset：电路复位标志；

sign：算法执行加密或者解密的标志；

roundsel：轮数选择标志，用于子密钥扩展模块；

ready：算法执行结束标志，在算法结束后给出 ready 为 1，若算法未结束，ready 为 0；

din：32 位接口变量，算法的输入和初始密钥共用；

dout：32 位的数据输出端口变量；

din1，din2，key1 以及 key2 是中间寄存器变量，寄存读入的数据和初始密钥；

dout1，dout2 是算法执行结束后的密文或明文寄存器变量（置换 IP^{-1} 的结果）。

各标志信号的高低电平选择可根据实际情况来设计，本书给出的定义仅供参考。

2.3 DES 算法工程实现

1．创建 ISE 工程

打开 ISE 开发环境，选择菜单 File→New Project，建立一个新工程，然后设置顶层文件模块名、存储目录和设计方式。接着选择器件，这里选择 Xilinx 公司的 Virtex5。再选择仿真工具、编程语言，这里选择第三方仿真软件 ModelSim 和 Verilog 语言。完成新工程的建立，如图 2.3 和图 2.4 所示。

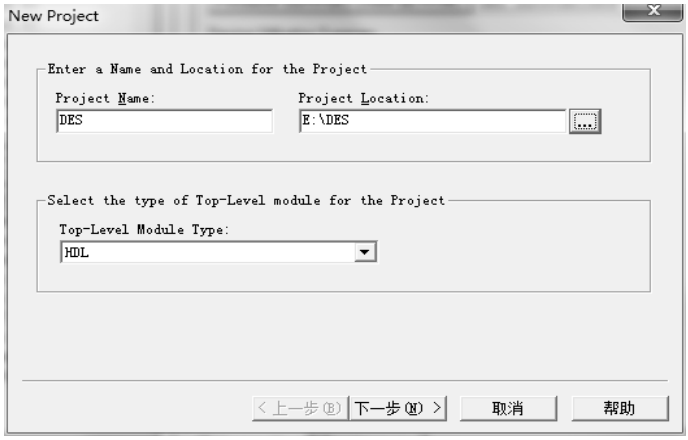

图 2.3 DES 算法工程文件的建立

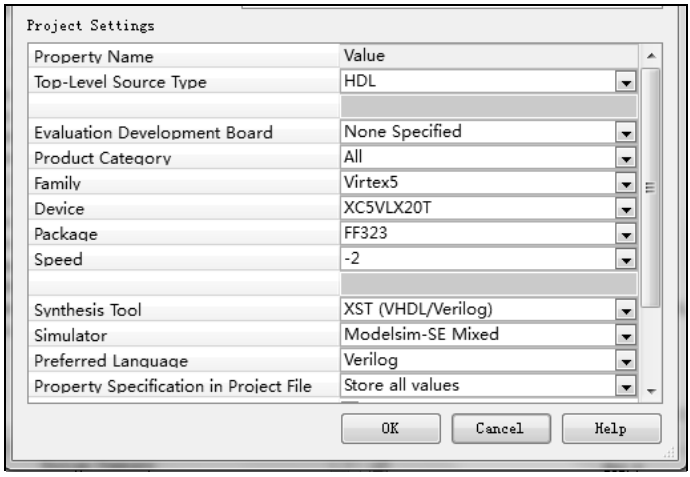

图 2.4 器件的选择和工具的使用

2．编写设计代码

将 2.2 节中编写的模块代码导入新建的工程中，具体实现如图 2.5 所示。

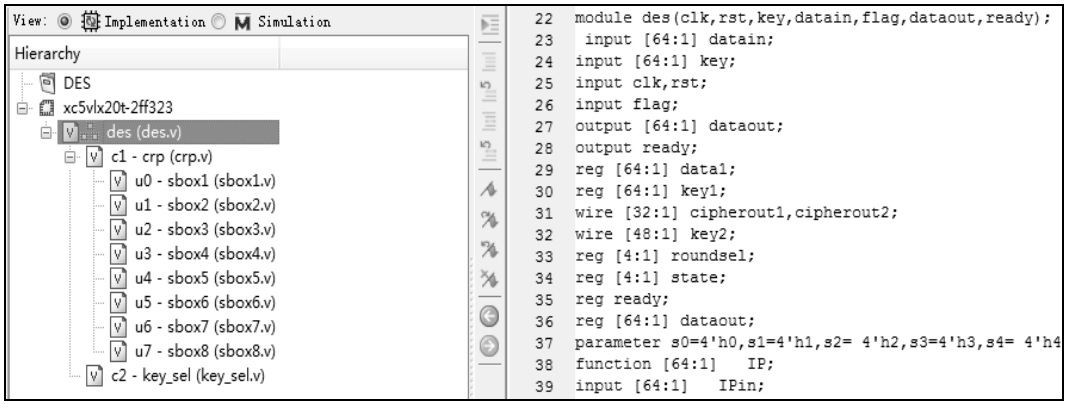

图 2.5 DES 算法的实现

3．工程验证

（1）根据以上编写的程序，编写该程序的测试文件，如图 2.6 所示。

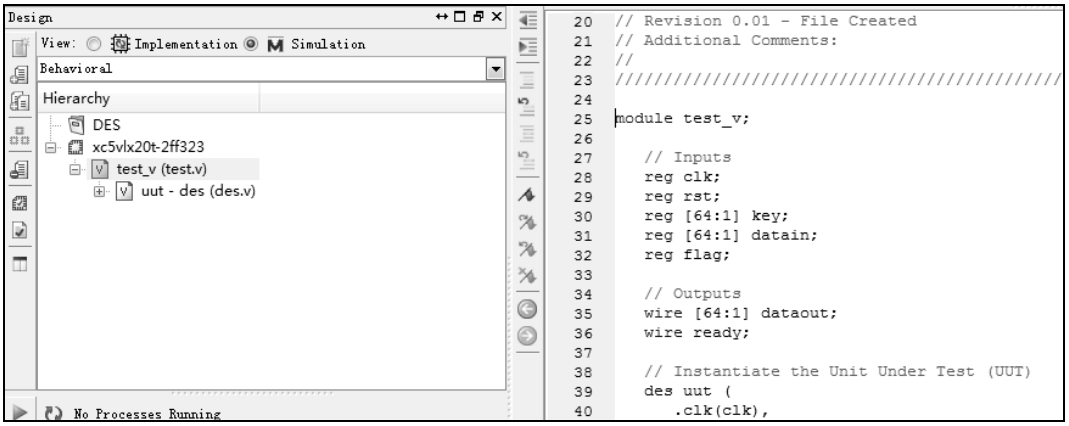

图 2.6　DES 算法测试文件

（2）根据上一步的测试文件，利用 ModelSim 仿真软件，对该算法进行仿真测试，仿真结果如图 2.7 和图 2.8 所示。

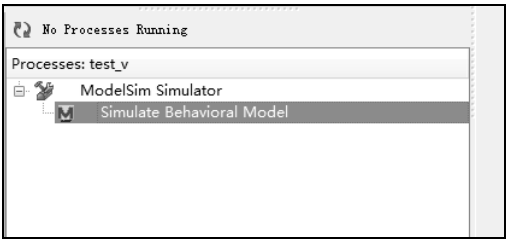

图 2.7　ModelSim 调用示意图

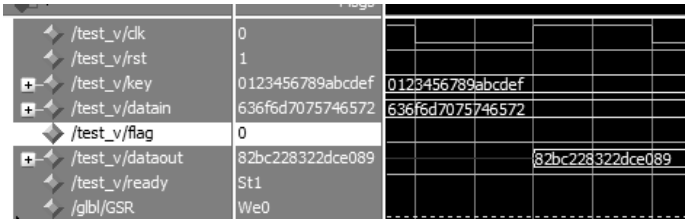

图 2.8　DES 算法加密仿真结果

（3）修改测试程序，解密得到明文，如图 2.9 所示，观察实验结果，验证 DES 解密算法。

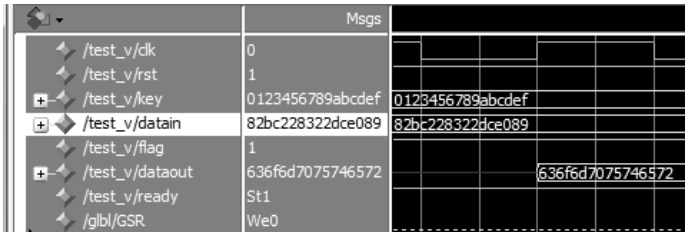

图 2.9　DES 算法解密仿真结果

2.4 效果测试

为了验证本 DES 算法设计的正确性，编写 test 测试程序，用 ModelSim SE10.1 仿真工具进行仿真，该测试所使用的计算机配置为：四核 i5-3470 CPU，4G 内存，32 位操作系统。

密钥：0123456789abcdef

明文：636f6d7075746572

得到的密文：82bc228322dce089

为了验证解密程序的正确性，采用相同密钥，将密文 82bc228322dce089 还原后，得到明文为 636f6d7075746572，从而进一步验证了设计的正确性。

加密所用时间为 260 ns，解密所用时间为 260 ns。

下面对该工程的时序、运行效率、资源占用情况进行分析。在 Xilinx 公司的 5vlx20tff323-2 FPGA 元器件上，使用 ISE 的 XST 分析工具进行综合，综合结果如图 2.10 所示。

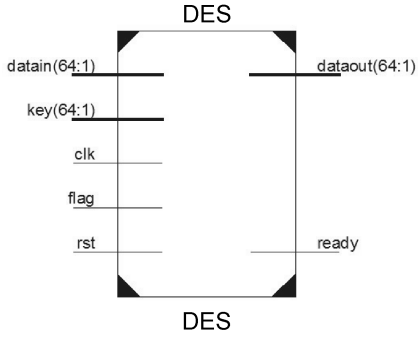

图 2.10　DES 算法综合后的外部器件

资源占用情况分析报告如表 2.1 所示。

表 2.1　DES 算法资源占用情况

Selected Device : 5vlx20tff323-2				
Number of Slice Registers	197	out of	12480	1%
Number of Slice LUTs	522	out of	12480	4%
Number of LUT Flip Flop pairs used	646			
Number of IOs	196			
Number of bonded IOBs	188	out of	172	109%
Number with an unused Flip Flop	449	out of	646	69%
Number with an unused LUT	124	out of	646	19%
Number of fully used LUT-FF pairs	73	out of	646	11%

时序分析报告如下。

Maximum Frequency: 251.664MHz

由报告可知，DES 算法的最高工作频率为 251.664 MHz，分析程序可知，对 64 个比特的

数据进行 DES 加解密操作需要用 33 个时钟。因此，DES 算法的最高处理速度为 64×251.664/33=488.076 Mbps。

2.5 本章小结

本章实现的 DES 算法，加密效率高，具有很高的实际应用价值。本章首先对 DES 的原理进行了详细的描述；然后用 Verilog 代码分别实现了 IP 和 IP^{-1} 模块、密钥扩展模块、S 盒模块、f 函数模块，并整合实现了 DES 的 FPGA 实现；最后通过功能仿真和代码检测，验证了代码功能的正确性。

DES 算法成熟已久，然而其密钥相对较短，已经不适应当今分布式开放网络对数据加密安全性的要求，但 DES 算法的设计理念深深影响了全世界对称密码算法的构造思路，至今仍然值得密码设计与密码工程应用人员关注和学习。

第 3 章　AES 算法 FPGA 实现

1997 年 4 月 15 日，美国国家标准技术研究所（National Institute of Standards and Technology，NIST）征集 AES（Advanced Encryption Standard）算法，以取代 DES 算法。1999 年 8 月，NIST 从 15 个算法中选出了 5 个 AES 候选算法，它们分别是 MARS、RC6、Rijndael、Serpent 和 Twofish。NIST 声称最后将从这 5 个候选算法中遴选一个或多个算法作为 AES 算法。2000 年 10 月 2 日，NIST 正式宣布 Rijndael 将被不加修改地作为 AES 算法。该算法为比利时密码学家 Joan Daemen 和 Vincent Rijmen 所设计，结合两位作者的名字，以 Rijndael 命名之。

3.1　AES 算法原理

AES 算法分组长度为 128bit，密钥长度可为 128bit/192bit/256bit。尽管人们对 AES 还有不同的看法，但总体来说，AES 作为新一代的数据加密标准，汇聚了强安全性、高性能、高效率、易用和灵活等优点。相对而言，AES 的 128 位密钥在综合性能方面比 DES 的 56 位密钥强约 1021 倍。AES 算法的加密和解密过程分别有 10 轮迭代，加密过程的每一轮迭代包括字节代换（ByteSubstitute）、行移位（ShiftRow）、列混合（MixColumn）、轮密钥加（RoundKey）四个步骤，但最后一轮没有列混合；解密过程的每一轮迭代包括行移位（ShiftRow）求逆、字节代换（ByteSubstitute）求逆、轮密钥加（RoundKey）求逆、列混合（MixColumn）求逆四个步骤，但最后一轮不包括列混合。种子密钥扩展为轮迭代所需要的 11 组子密钥。

3.1.1　基础知识

AES 中的运算一部分是按字节（有限域 $GF(2^8)$ 的元素）定义的，另一部分是按 4 个字节组成的字定义的，一个字为 32 位，可看成是系数在有限域 $GF(2^8)$ 上的次数小于 4 的多项式。有限域 $GF(2^8)$ 是由不可约多项式 $m(x)=x^8+x^4+x^3+x+1$（对应二进制 100011011，十六进制 11）定义的，其中的元素可以表示成 $GF(2^8)$ 上的多项式形式，也可以表示成字节的形式，还可以表示成十六进制形式。例如，一个由 01010111 组成的字节可以表示成多项式 $x^6+x^4+x^2+x+1$，也可以是十六进制 57。在实际的操作过程中，视具体情况采用相应的表示方法。这里所选的 $m(x)$ 是所有次数为 8 的不可约多项式列表中的一个。AES 算法用到 $GF(2^8)$ 域的运算，该域中的加法定义为简单的 bit 位异或，乘法运算相对复杂一些。

用 X 乘以一个多项式简称为 X 乘。由此得出 X（十六进制表示为02）乘可以用字节内左移一位和紧接着的一个与十六进制数 1B 的按位模 2 加来实现，该计算记为 X time() 。X 乘运算定义如下：

$$X\ \text{time}(\underline{b})=\begin{cases}\underline{b}<<1, & \text{if } b_7=0\\ (\underline{b}<<1)\oplus 0\text{x1B}, & \text{if } b_7=1\end{cases}$$

3.1.2　加密解密过程

下面介绍轮密钥加的过程。

轮密钥加是将 128 比特密钥 K_i 同状态中的数据进行逐位异或操作，该过程可以看成是字逐位异或的结果，也可以看成字节级别的操作。其中，密钥 K_i 中每个字 $w[4i], w[4i+1], w[4i+2], w[4i+3]$ 均为 32 bit，包含 4 个字节。密钥 K_i（$i=0,1,\cdots,10$）的生成过程为密钥扩展算法。AES 的密钥长度和加密轮数如表 3.1 所示。

表 3.1　AES 算法的密钥长度和加密轮数列表

	密钥长度（32 比特字）	分组长度（32 比特字）	加密轮数
AES-128	4	4	10
AES-192	6	4	12
AES-256	8	4	14

AES 算法的加密与解密流程图如图 3.1 所示。

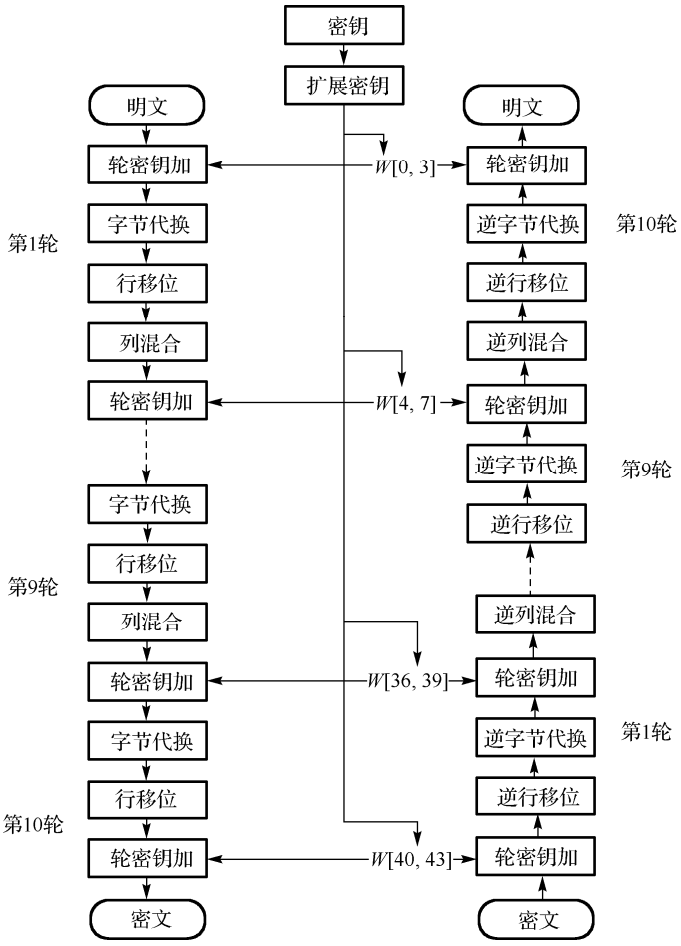

图 3.1　AES 算法加密解密流程图

AES 加密的第 1 轮到第 9 轮的轮函数是一样的，包括 4 个操作：字节代换、行移位、列

混合和轮密钥加。最后一轮迭代不执行列混合。另外，在第 1 轮迭代之前，先将明文和原始密钥进行一次加密操作。

AES 的解密过程仍为 10 轮，每一轮的操作是加密操作的逆操作。由于 AES 的 4 个轮操作（字节代换、行移位、列混合和轮密钥加）都是可逆的，因而解密操作的一轮就是顺序执行逆行移位，逆字节代换、轮密钥加和逆列混合。同加密操作类似，最后一轮不执行逆列混合，在第 1 轮解密之前，要执行 1 次轮密钥加操作。

3.2 AES 算法相关模块 FPGA 设计

本章 AES 算法的 FPGA 设计采用 Verilog HDL 语言，这里给出了程序设计框架，如图 3.2 所示。AES 算法的 FPGA 设计执行模块化设计思路，包括轮密钥加变换模块、字节代换模块、密钥扩展模块、行移位模块以及列混合模块。与 DES 算法不同，AES 算法在加解密过程中需要使用不同的 S 盒（S 盒，逆 S 盒）以及不同的轮函数，因此程序模块需要分别设计。

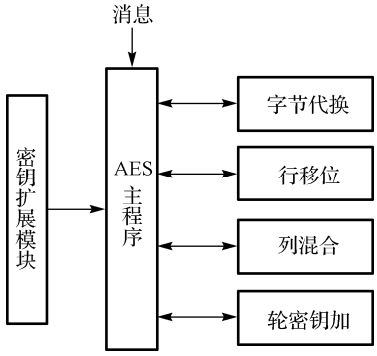

图 3.2 AES 算法加密程序设计框架

3.2.1 密钥加变换设计

轮密钥加变换又称为密钥加变换，128 位的 State 按位与 128 位的密钥 XOR：$(b_{0j},b_{1j},b_{2j},b_{3j}) \leftarrow (b_{0j},b_{1j},b_{2j},b_{3j}) \oplus (k_{0j},k_{1j},k_{2j},k_{3j})$，对 $j=0,\cdots,9$ 轮密钥加变换虽然简单，却影响了 State 中的每一位。密钥扩展的复杂性和 AES 其他阶段运算的复杂性，确保了该算法的安全性。

3.2.2 字节代换模块设计

字节代换是非线性变换，独立地对状态的每个字节进行。当需要对某个字进行变换时，将该字的前两个字节代换为表 3.2 中对应的行数，后两个字节代换为对应的列数，然后查表便可找出对应的字变换。可以将上述字节代换（Byte Substitute）做成代换表（S 盒），如表 3.2 所示。

AES 定义的 S 盒中，State 中每个字节按照如下方式映射为一个新的字节：把该字节的高 4 位作为行值，低 4 位作为列值，然后取出 S 盒中对应行和列的元素作为输出。例如，十六进制数 0x84，对应 S 盒的行是 8，列是 4，S 盒中该位置对应的值是 0x5F。

表 3.2　字节代换表（S 盒）

行＼列	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	63	7C	77	7B	F2	6B	6F	C5	30	01	67	2B	FE	D7	AB	76
1	CA	82	C9	7D	FA	59	47	F0	AD	D4	A2	AF	9C	A4	72	C0
2	B7	FD	93	26	36	3F	F7	CC	34	A5	E5	F1	71	D8	31	15
3	04	C7	23	C3	18	96	05	9A	07	12	80	E2	EB	27	B2	75
4	09	83	2C	1A	1B	6E	5A	A0	52	3B	D6	B3	29	E3	2F	84
5	53	D1	00	ED	20	FC	B1	5B	6A	CB	BE	39	4A	4C	58	CF
6	D0	EF	AA	FB	43	4D	33	85	45	F9	02	7F	50	3C	9F	A8
7	51	A3	40	8F	92	9D	38	F5	BC	B6	DA	21	10	FF	F3	D2
8	CD	0C	13	EC	5F	97	44	17	C4	A7	7E	3D	64	5D	19	73
9	60	81	4F	DC	22	2A	90	88	46	EE	B8	14	DE	5E	0B	DB
A	E0	32	3A	0A	49	06	24	5C	C2	D3	AC	B2	91	95	E4	79
B	E7	C8	37	6D	8D	D5	4E	A9	6C	56	F4	EA	65	7A	AE	08
C	BA	78	25	2E	1C	A6	B4	C6	E8	DD	74	1F	4B	BD	8B	8A
D	70	3E	98	11	48	03	F6	0E	61	35	57	B9	86	C1	1D	9E
E	E1	F8	98	11	69	D9	8E	94	9B	1E	87	E9	CE	55	28	DF
F	8C	A1	89	0D	BF	E6	42	68	41	99	2D	0F	B0	54	BB	16

S 盒是一个由 16×16 字节组成的矩阵，包含了 8 位值所能表达的 256 种可能的变换。S 盒按照以下方式构造。

（1）逐行按照升序排列的字节值初始化 S 盒。第一行是{00}，{01}，{02}，…，{0F}；第二行是{10}，{l1}，…，{1F}等。在行 X 和列 Y 的字节值是{xy}。

（2）把 S 盒中的每个字节映射为它在有限域 GF(2^8) 中的逆，{00}被映射为它自身{00}。GF(2^8) 由一组从 0x00 到 0xFF 的 256 个值组成，加上加法和乘法运算。

（3）把 S 盒中的每个字节记成 $(b_8,b_7,b_6,b_5,b_4,b_3,b_2,b_1,b_0)$。对 S 盒中每个字节的每位做如下变换：

$$b_i' = b_i \oplus b_{(i+4)\bmod 8} \oplus b_{(i+5)\bmod 8} \oplus b_{(i+6)\bmod 8} \oplus b_{(i+7)\bmod 8} \oplus c_i$$

上式中 c_i 是指值为 0x63 的字节 C 的第 i 位，即 $(c_8,c_7,c_6,c_5,c_4,c_3,c_2,c_1,c_0)=(01100011)$。符号（$b_i'$）表示更新后的变量的值。AES 用下面的矩阵方式描述了这个变换。

$$\begin{bmatrix} b_0' \\ b_1' \\ b_2' \\ b_3' \\ b_4' \\ b_5' \\ b_6' \\ b_7' \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

本程序主要通过字节代换公式来设计字节代换程序，设计的 S 盒如程序 3-1 所示。

程序 3-1：

```
first_mux_aA=first_mux_InvInput[1]^first_mux_InvInput[7];
first_mux_aB=first_mux_InvInput[5]^first_mux_InvInput[7];
first_mux_aC=first_mux_InvInput[4]^first_mux_InvInput[6];
first_mux_al_t[0]=first_mux_aC^first_mux_InvInput[0]^first_mux_InvInput[5];
first_mux_al_t[1]=first_mux_InvInput[1]^first_mux_InvInput[2];
first_mux_al_t[2]=first_mux_aA;
first_mux_al_t[3]=first_mux_InvInput[2]^first_mux_InvInput[4];
first_mux_ah_t[0]=first_mux_aC^first_mux_InvInput[5];
first_mux_ah_t[1]=first_mux_aA^first_mux_aC;
first_mux_ah_t[2]=first_mux_aB^first_mux_InvInput[2]^first_mux_InvInput[3];
first_mux_ah_t[3]=first_mux_aB;
al = (first_mux_al_t);
ah = (first_mux_ah_t);
next_ah_reg = (first_mux_ah_t);
……
reg[7:0] end_mux_data_var,end_mux_data_o_var;
reg end_mux_aA,end_mux_aB,end_mux_aC,end_mux_aD;
always @(decrypt_i or inva)
begin
   end_mux_data_var=inva;
    case(decrypt_i)
     0:
     begin
     end_mux_aA=end_mux_data_var[0]^end_mux_data_var[1];end_mux_aB=end_
     mux_data_var[2]^end_mux_data_var[3];
     end_mux_aC=end_mux_data_var[4]^end_mux_data_var[5];end_mux_aD=end_
     mux_data_var[6]^end_mux_data_var[7];
  end_mux_data_o_var[0]=(!end_mux_data_var[0])^end_mux_aC^end_mux_aD;
  end_mux_data_o_var[1]=(!end_mux_data_var[5])^end_mux_aA^end_mux_aD;
  end_mux_data_o_var[2]=end_mux_data_var[2]^end_mux_aA^end_mux_aD;
  end_mux_data_o_var[3]=end_mux_data_var[7]^end_mux_aA^end_mux_aB;
  end_mux_data_o_var[4]=end_mux_data_var[4]^end_mux_aA^end_mux_aB;
  end_mux_data_o_var[5]=(!end_mux_data_var[1])^end_mux_aB^end_mux_aC;
  end_mux_data_o_var[6]=(!end_mux_data_var[6])^end_mux_aB^end_mux_aC;
  end_mux_data_o_var[7]=end_mux_data_var[3]^end_mux_aC^end_mux_aD;
  data_o = (end_mux_data_o_var);
     end
    default:
     begin
       data_o = (end_mux_data_var);
     end
```

```
    endcase
end
```

AES 算法中逆 S 盒设计方式同程序 3-1 类似，根据表 3.1 修改不同的分支语句即可。

如果使用 case 语言来设计 AES 算法的 S 盒以及逆 S 盒，将占用大量硬件逻辑资源。为节省芯片的逻辑资源，在芯片存储器容量允许的情况下，可以使用芯片的存储器来设计 S 盒以及逆 S 盒，提高芯片资源的利用率。

3.2.3　密钥扩展模块设计

为了防止已有的密码分析攻击，AES 使用了与轮相关的轮常量（Rcon[*j*]）来防止不同轮中产生的轮密钥的对称性或相似性。轮常量是一个字，这个字的右边三个字节总为 0，如表3.3 所示。AES 在加密和解密算法中使用了一个由种子密钥字节数组生成的密钥调度表，AES 规范中称之为密钥扩展（Key Expansion）。密钥扩展例程从一个原始密钥中生成多重密钥以代替使用单个密钥，大大增加了比特位的扩散，AES 密钥扩展算法的输入值是 4 字密钥，输出是一个 44 字的一维线性数组，这足以为初始轮密钥扩展阶段和算法中的其他 10 轮中的每一轮提供 16 字节的轮密钥。

通过生成器产生 N_r+1 轮密钥，每个轮密钥由 N_b 个字组成，共有 $N_b(N_r+1)$ 个字 $W[i], i=0,1,\cdots, N_b(N_r+1)-1$。

在加密过程中，需要 N_r+1 个子密钥，需要构造 $4(N_r+1)$ 个 32 位字。

表 3.3　Rcon[*j*]数据表格

j	1	2	3	4	5
Rcon[*j*]	01000000	02000000	04000000	08000000	10000000
j	6	7	8	9	10
Rcon[*j*]	20000000	40000000	80000000	1b000000	36000000

输入密钥直接被复制到扩展密钥数组的前四个字中，得到 *w*[0]、*w*[1]、*w*[2]、*w*[3]；然后每次用四个字填充扩展密钥数组余下的部分。在扩展密钥数组中，*w*[*i*]的值依赖于 *w*[*i*–1]和 *w*[*i*–4]（$i \geqslant 4$）。

对 *w* 数组中下标不为 4 的倍数的元素，只是简单地异或，其逻辑关系为：$w[i]=w[i-1]\oplus w[i-4]$（*i* 不为 4 的倍数）。

对 *w* 数组中下标为 4 的倍数的元素，采用如下的计算方法。

（1）将一个字的四个字节循环左移一个字节，即将字 $[b_0,b_1,b_2,b_3]$ 变为 $[b_0',b_1',b_2',b_3']$；

（2）基于 S 盒对输入字中的每个字节进行 S 代替；

（3）将步骤（1）和步骤（2）的结果再与轮常量 Rcon[*j*]相异或。

程序 3-2：

通过 case 语句来给 Rcon[*j*]赋值，例如给 Rcon[0]赋值语句为

```
case(Round_i)//判断 i 值
   1:
    begin
      Rcon_o = (1); //给 Rcon[j]赋值
```

```
    end
always @(start_i or last_key_i or sbox_data_i or state or rcon_o or col
        or key_reg)
begin
  K_var=last_key_i;
  case(state)
    0:
     begin
       if(start_i)
       begin
         col_t=0;
         sbox_access_o = (1);
         sbox_data_o = (K_var[31:24]);
         next_state = (1);
       end
     end
    1:
     begin
       sbox_access_o = (1);
       sbox_data_o = (K_var[23:16]);
       col_t[7:0]=sbox_data_i;
       next_col = (col_t);
       next_state = (2);
     end
    2:
     begin
       sbox_access_o = (1);
       sbox_data_o = (K_var[15:8]);
       col_t[31:24]=sbox_data_i;
       next_col = (col_t);
       next_state = (3);
     end
    3:
     begin
       sbox_access_o = (1);
       sbox_data_o = (K_var[7:0]);
       col_t[23:16]=sbox_data_i;
       next_col = (col_t);
       next_state = (4);
     end
    4:
     begin
       sbox_access_o = (1);
       col_t[15:8]=sbox_data_i;
```

```
            next_col = (col_t);
            W_var[127:96]=col_t^K_var[127:96]^{rcon_o,zero};
            W_var[95:64]=W_var[127:96]^K_var[95:64];
            W_var[63:32]=W_var[95:64]^K_var[63:32];
            W_var[31:0]=W_var[63:32]^K_var[31:0];
            next_ready_o = (1);
            next_key_reg = (W_var);
            next_state = (0);
        end
```

通过上述程序就能实现密钥扩展设计者的思想，完成密钥扩展功能。

3.2.4　行移位设计

行移位设计主要是保障 State 的第 1 行字节保持不变，State 的第 2 行字节循环左移 1 个字节，State 的第 3 行字节循环左移 2 个字节，State 的第 4 行循环左移 3 个字节。行移位变换如图 3.3 所示，其对应的程序实现如程序 3-3 所示。

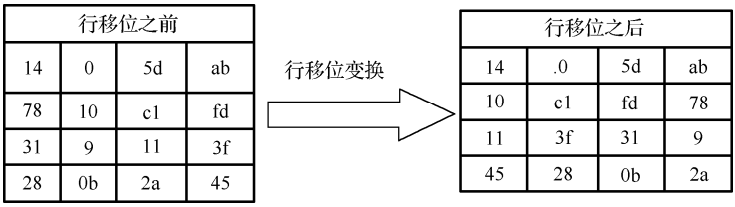

图 3.3　行移位变换

程序 3-3：

```
    data_reg_128[127:120]=data_reg_var[0];
    data_reg_128[119:112]=data_reg_var[13];
    data_reg_128[111:104]=data_reg_var[10];
    data_reg_128[103:96]=data_reg_var[7];
    data_reg_128[95:88]=data_reg_var[4];
    data_reg_128[87:80]=data_reg_var[1];
    data_reg_128[79:72]=data_reg_var[14];
    data_reg_128[71:64]=data_reg_var[11];
    data_reg_128[63:56]=data_reg_var[8];
    data_reg_128[55:48]=data_reg_var[5];
    data_reg_128[47:40]=data_reg_var[2];
    data_reg_128[39:32]=data_reg_var[15];
    data_reg_128[31:24]=data_reg_var[12];
    data_reg_128[23:16]=data_reg_var[9];
    data_reg_128[15:8]=data_reg_var[6];
    data_reg_128[7:0]=data_reg_var[3];
```

本程序将数据输入以一个字节为单位进行整合交换，达到行移位效果。

3.2.5 列混合设计

列混合变换是一个替代操作，是 AES 最具技巧性的部分，该步骤的设计准则主要有四个方面：维数、线性性、扩散性和在 8 位处理器上的性能。它只在 AES 的第 0，1，…，N_r–1 轮中使用，在第 N_r 轮中不使用该变换。乘积矩阵中的每个元素都是一行和一列对应元素的乘积之和。在 MixColumns 变换中，乘法和加法都是定义在 GF(2^8) 上的。

列混合变换写成矩阵形式如下。

$$\begin{bmatrix} s'_{0,0} & s'_{0,1} & s'_{0,2} & s'_{0,3} \\ s'_{1,0} & s'_{1,1} & s'_{1,2} & s'_{1,3} \\ s'_{2,0} & s'_{2,1} & s'_{2,2} & s'_{2,3} \\ s'_{3,0} & s'_{3,1} & s'_{3,2} & s'_{3,3} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1,0} & s_{1,1} & s_{1,2} & s_{1,3} \\ s_{2,0} & s_{2,1} & s_{2,2} & s_{2,3} \\ s_{3,0} & s_{3,1} & s_{3,2} & s_{3,3} \end{bmatrix}$$

列混合过程用到有限域 GF(2^8) 乘法，其中，任何值乘 0x01 都等于其自身；用 0x02 做乘法时可以使用乘法运算 Xtime( )来描述；用 0x03 做乘法时，可以采用 b∗0x03 = b∗(0x02⊕0x01) = (b∗0x02)⊕(b∗0x01)，这里∗表示 GF(2^8) 乘法，⊕表示异或操作。

逆向列混合变换可由下式的矩阵乘法定义，并且可以验证逆变换矩阵同正变换矩阵的乘积恰好为单位矩阵。

$$\begin{bmatrix} s'_{0,0} & s'_{0,1} & s'_{0,2} & s'_{0,3} \\ s'_{1,0} & s'_{1,1} & s'_{1,2} & s'_{1,3} \\ s'_{2,0} & s'_{2,1} & s'_{2,2} & s'_{2,3} \\ s'_{3,0} & s'_{3,1} & s'_{3,2} & s'_{3,3} \end{bmatrix} = \begin{bmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{bmatrix} \begin{bmatrix} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1,0} & s_{1,1} & s_{1,2} & s_{1,3} \\ s_{2,0} & s_{2,1} & s_{2,2} & s_{2,3} \\ s_{3,0} & s_{3,1} & s_{3,2} & s_{3,3} \end{bmatrix}$$

列混合主要程序设计如程序 3-4 所示。

程序 3-4：

```
always @(decrypt_i or start_i or state or data_reg or outmux or data_o_reg
        or data_i)
begin
  data_i_var=data_i;
  data_reg_var=data_reg;
  next_data_reg = (data_reg);
  next_state = (state);
  mix_word = (0);
  next_ready_o = (0);
  next_data_o = (data_o_reg);
  case(state)
    0:
      begin
        if(start_i)
        begin
          aux=data_i_var[127:96];
          mix_word = (aux);
          data_reg_var[127:96]=outmux;
```

```
            next_data_reg = (data_reg_var);
            next_state = (1);
          end
        end
      1:
        begin
          aux=data_i_var[95:64];
          mix_word = (aux);
          data_reg_var[95:64]=outmux;
          next_data_reg = (data_reg_var);
          next_state = (2);
        end
      2:
        begin
          aux=data_i_var[63:32];
          mix_word = (aux);
          data_reg_var[63:32]=outmux;
          next_data_reg = (data_reg_var);
          next_state = (3);
        end
      3:
        begin
          aux=data_i_var[31:0];
          mix_word = (aux);
          data_reg_var[31:0]=outmux;
          next_data_o = (data_reg_var);
          next_ready_o = (1);
          next_state = (0);
        end
      default:
        begin
        end
    endcase
```

本程序实现了列混合变换，该步骤的设计准则主要有 4 个方面：维数、线性性、扩散性和在 8 位处理器上的性能。它在 AES 的第 $0,1,\cdots,N_r-1$ 轮中使用，乘积矩阵中的每个元素都是一行和一列对应元素的乘积之和。

3.3 AES 算法工程实现

1．创建 ISE 工程

打开 ISE 开发环境，选择菜单 File→New Project，建立一个新工程，然后设置顶层文件模块名、存储目录和设计方式。接着选择器件，这里选择 Xilinx 的 Virtex4 来实现，再选择第三方仿真软件 ModelSim 和 Verilog 语言进行仿真测试。完成新工程的建立，如图 3.4 和图 3.5 所示。

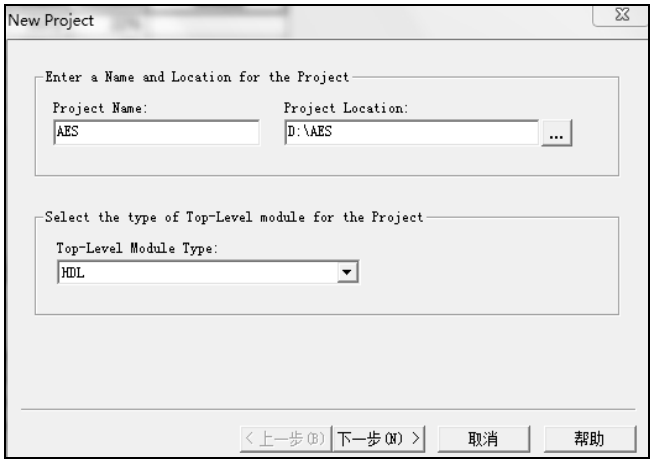

图 3.4 建立 AES 工程文件

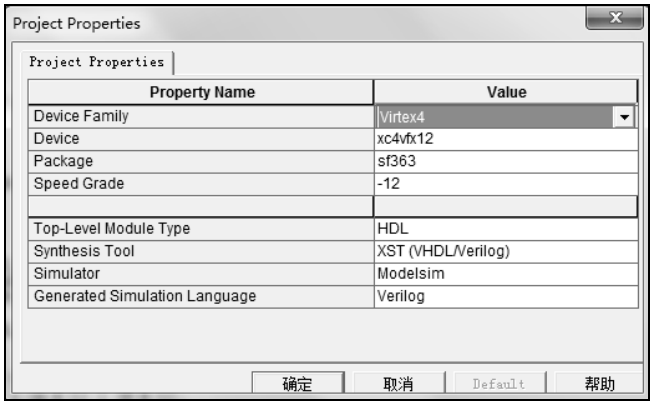

图 3.5 建立 AES 工程文件属性设置图

2．编写设计代码

将 3.2 节中编写的模块代码导入新建的工程中，具体实现如图 3.6 所示。

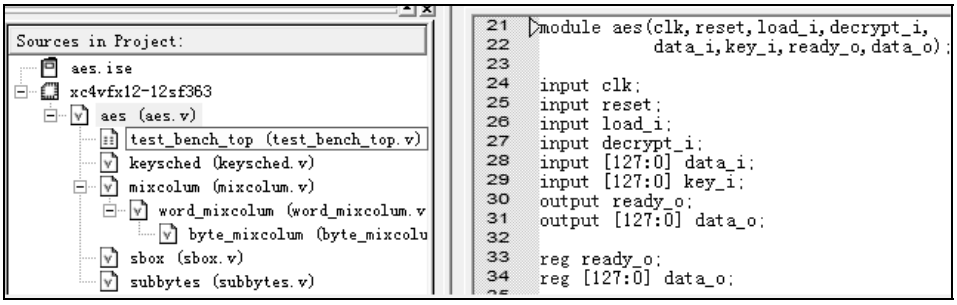

图 3.6 AES 算法的实现

3．工程验证

（1）根据以上编写的程序，编写该程序的测试文件，如图 3.7 所示。

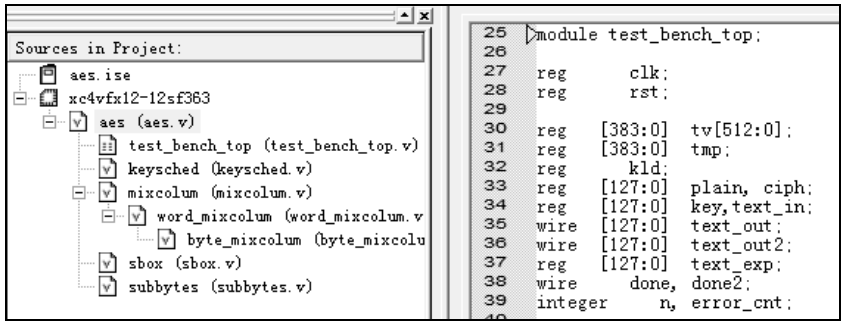

图 3.7　AES 算法测试文件

（2）根据上一步的测试文件，利用 ModelSim 仿真软件，对该算法进行仿真测试，如图 3.8 所示。仿真结果如图 3.9 和图 3.10 所示。

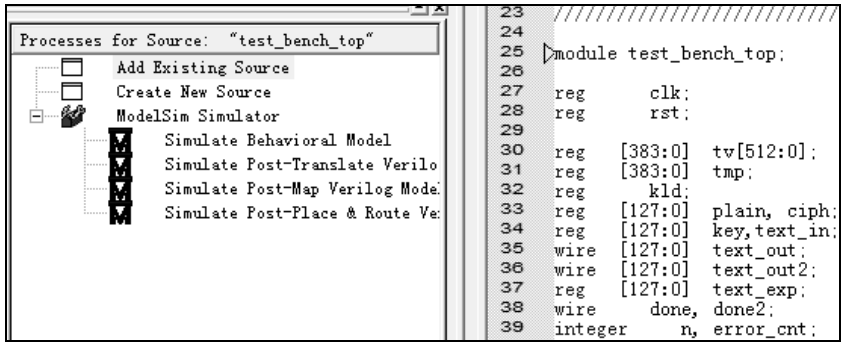

图 3.8　ModelSim 调用示意图

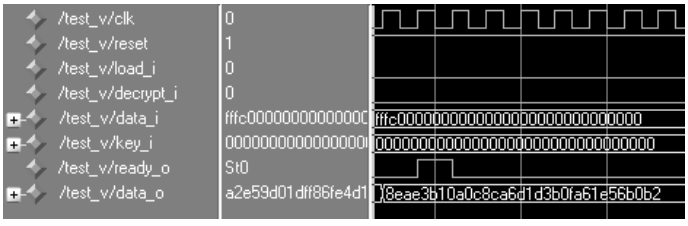

图 3.9　AES 算法加密仿真结果

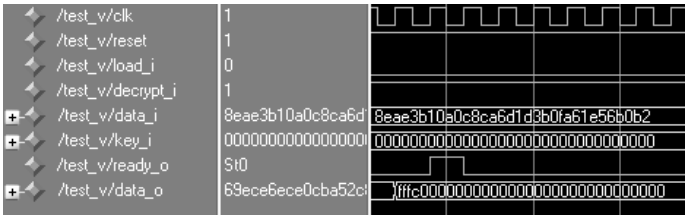

图 3.10　AES 算法解密仿真结果

3.4　效果测试

通过功能仿真与系统测试，可以验证整个设计的正确性，为了测试其性能，还需要对整个设计在芯片上的综合结果进行分析。

仿真时，在波形文件中需要给出合适的时钟周期。在 FPGA 程序设计中，验证算法的正确性和设计合理性的方法之一是对程序进行波形仿真。在本书中使用 ModelSim 对算法进行仿真，以验证其功能是否正确。例如使用明文（16 进制）为 fffc0000000000000000000000000000，密钥为 00000000000000000000000000000000（读者可自行验证，将其设为其他值，也可进行加解密操作），加密结果为（16 进制）8eae3b10a0c8ca6d1d3b0fa61e56b0b2。设置加解密标志位 decrypt_i 为 1，可以进行解密操作。将上述加密结果作为输入，密钥不变，可以得到明文为：fffc0000000000000000000000000000，进一步验证了算法 FPGA 实现的正确性。

下面为对该工程的时序、运行效率、资源占用情况进行分析。在 Xilinx 公司的 4vfx12sf363-12 元器件上综合，经过 ISE 的 XST 分析工具进行综合。综合结果如图 3.11 所示。

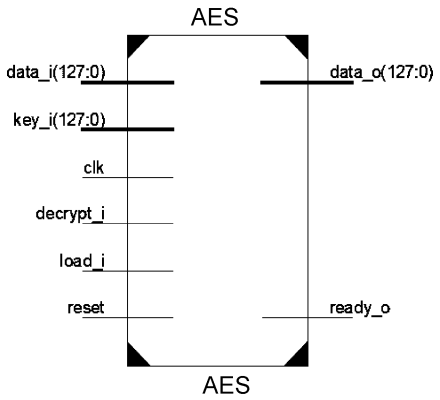

图 3.11　AES 算法综合后的外部器件

资源占用情况分析报告如表 3.4 所示。

表 3.4　AES 算法资源占用情况

Selected Device : 4vfx12sf363-12	
Number of Slices	675 out of 12480 5%
Number of Slice Flip Flops	857 out of 1532 55%
Number of 4 input LUTs	1425 out of 12480 11%
Number of IOs	389
Number of bonded IOBs	389 out of 172 26%
IOB Flip Flops	1
Number of GCLKs	1 out of 32 3%

时序分析报告如下。

Timing constraint: Default period analysis for Clock 'clk'

Clock period: 5.700ns (frequency: 175.452MHz)

Total number of paths / destination ports: 189723 / 931

由时序分析报告结果可知，AES 算法的最高工作频率为 175.452 MHz。分析程序可知，处理 128 比特的分组数据共用 505 个时钟。所以，AES 算法的最高处理速度计算结果为 128×175.452/505=44 Mbps。

由以上分析报告可知，工程是在 Xilinx 公司的 4vfx12sf363-12 FPGA 元器件上实现，芯片

上使用的查找表数为 1425 个，占芯片资源的 11%；使用了 389 个引脚，占引脚资源的 26%。从中可以看出使用的逻辑元器件并不多，很好地实现了性能与效率的折中。

3.5 本章小节

随着对称密码的发展，DES 数据加密标准算法由于密钥长度较小（56 位），已经不适应当今分布式开放网络对数据加密安全性的要求。因此，1997 年 NIST 公开征集新的数据加密标准，即 AES。经过三轮的筛选，比利时的 Joan Daeman 和 Vincent Rijmen 提交的 Rijndael 算法被提议为 AES 的最终算法。

本章主要对 AES 算法及其 FPGA 硬件实现进行了介绍。首先介绍了 AES 算法的原理与数学基础，使读者大致了解 AES 算法。接着重点介绍了 AES 算法的 FPGA 实现方法，包括 S 盒设计、密钥扩展和轮函数设计等几个方面，并给出了关键部分的设计代码。最后给出了仿真结果并对资源占用情况与工作效率进行了分析，使读者进一步加深了对 AES 算法 FPGA 实现过程的理解。

第 4 章　SM4 算法 FPGA 实现

SM4 是我国公布的主要用于无线局域网产品中的分组密码算法，SM4 分组长度和密钥长度均为 128bit，加密算法与密钥扩展算法都采用 32 轮非线性迭代结构，其中非线性变换中所使用的 S 盒是一个 8bit 输入 8bit 输出的置换。

4.1　SM4 算法原理

4.1.1　算法定义

（1）S 盒

S 盒为固定的 8 bit 输入 8 bit 输出的置换，记为 Sbox。

（2）基本运算

在 SM4 算法中采用了以下基本运算：

⊕　32 bit 异或；

<<<i　32 bit 循环左移 i 位。

（3）密钥及密钥参量

加密密钥长度为 128 bit，表示为 $\mathrm{MK}=(\mathrm{MK}_0,\mathrm{MK}_1,\mathrm{MK}_2,\mathrm{MK}_3)$，其中 $\mathrm{MK}_i(i=0,1,\cdots,31)$ 为字。轮密钥表示为 $(\mathrm{rk}_0,\mathrm{rk}_1,\cdots,\mathrm{rk}_{31})$，其中 $\mathrm{rk}_i(i=0,1,\cdots,31)$ 为字。轮密钥由加密密钥生成。$\mathrm{FK}=(\mathrm{FK}_0,\mathrm{FK}_1,\mathrm{FK}_2,\mathrm{FK}_3)$ 为系统参数，$\mathrm{CK}=(\mathrm{CK}_0,\mathrm{CK}_1,\cdots,\mathrm{CK}_{31})$ 为固定参数，用于密钥扩展算法，其中 FK_i $(i=0,1,\cdots,3)$，$\mathrm{CK}_i(i=0,1,\cdots,31)$ 为字。

4.1.2　算法描述

SM4 算法采用非线性迭代结构，以字（表示为 Z_2^{32}）为单位进行加密运算，称一次迭代运算为一轮变换。

设输入为 $(X_0,X_1,X_2,X_3)\in(Z_2^{32})^4$，轮密钥为 $\mathrm{rk}\in Z_2^{32}$，则轮变换 F 为

$$F(X_0,X_1,X_2,X_3,\mathrm{rk})=X_0\oplus T(X_1\oplus X_2\oplus X_3\oplus \mathrm{rk})$$

合成置换 T：

$Z_2^{32}\to Z_2^{32}$ 是一个可逆变换，由非线性变换 τ 和线性变换 L 复合而成，即 $T(.)=L(\tau(.))$。

非线性变换 τ：

τ 由 4 个并行的 S 盒构成，S 盒为固定的 8bit 输入 8bit 输出的置换，记为 Sbox(·)。

设输入为 $A=(a_0,a_1,a_2,a_3)\in(Z_2^8)^4$，输出为 $B=(b_0,b_1,b_2,b_3)\in(Z_2^8)^4$，则

$$(b_0,b_1,b_2,b_3)=\tau(A)=(\mathrm{Sbox}(a_0),\mathrm{Sbox}(a_1),\mathrm{Sbox}(a_2),\mathrm{Sbox}(a_3))$$

线性变换 L：

非线性变换 τ 的输出是线性变换 L 的输入。设输入为 $B \in Z_2^{32}$，输出为 $C \in Z_2^{32}$，则

$$C = L(B) = B \oplus (B <<< 2) \oplus (B <<< 10) \oplus (B <<< 18) \oplus (B <<< 24)$$

4.1.3　加解密算法

加密流程如图 4.1 所示，加密算法描述如下。

设明文输入为：$(X_0, X_1, X_2, X_3) \in (Z_2^{32})^4$，密文输出为：$(Y_0, Y_1, Y_2, Y_3) \in (Z_2^{32})^4$，子密钥为：$\mathrm{rk}_0, \mathrm{rk}_1, \cdots, \mathrm{rk}_{31} \in Z_2^{32}$，则

$$X_{i+4} = F(X_i, X_{i+1}, X_{i+2}, X_{i+3}, rk_i) = X_i \oplus T(X_{i+1} \oplus X_{i+2} \oplus X_{i+3} \oplus \mathrm{rk}_i), \quad i = 0,1,\cdots,31$$

$$(Y_0, Y_1, Y_2, Y_3) = (X_{35}, X_{34}, X_{33}, X_{32})$$

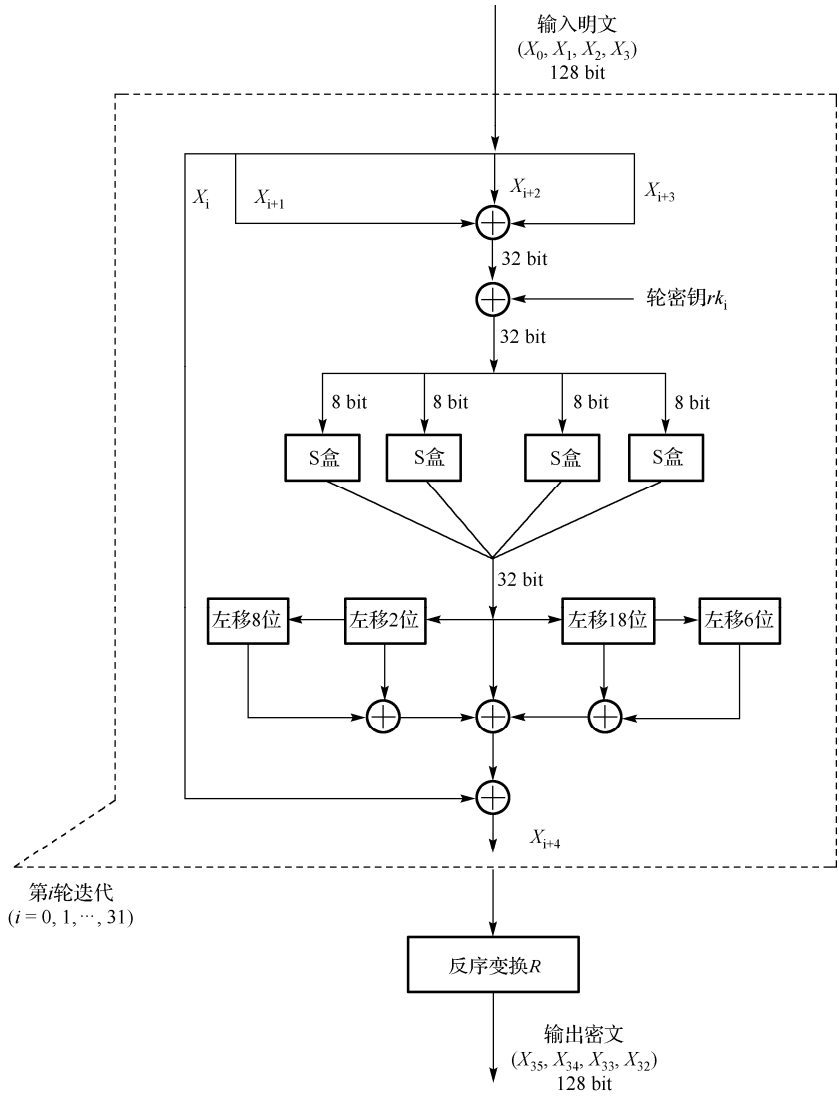

图 4.1　SM4 算法的加密流程

其中，以 32 位为单位的逆序输出是为了加解密的一致性。解密算法与加密算法的结构相似，只是轮密钥的使用顺序相反。

加密时轮密钥的使用顺序为：$rk_0, rk_1, \cdots, rk_{31}$。

解密时轮密钥的使用顺序为：$rk_{31}, rk_{30}, \cdots, rk_0$。

4.2 SM4 算法相关模块 FPGA 设计

通过对算法结构的分析发现，实现该算法需要完成四个模块：循环移位设计、S 盒设计、密钥扩展设计和轮函数加密设计。SM4 算法的 FPGA 模块设计结构如图 4.2 所示。

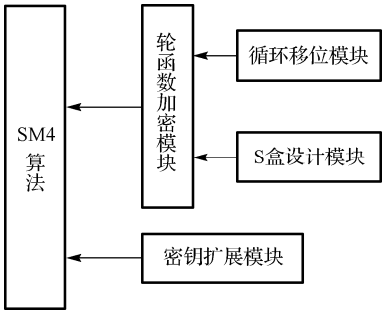

图 4.2 SM4 算法硬件设计结构

4.2.1 循环移位设计

在密钥扩展设计、轮函数加密中均会用到循环移位设计。循环移位打乱了数据顺序，增加了破译难度。例如：设计的左循环移位 2 位即 $B <<< 2$，如程序 4-1 所示。

程序 4-1：

```
module B_SHIFT1 (b1_out,
                 b1_in
                 );
parameter BWIDTH=32;
output [0:BWIDTH-1] b1_out;
reg    [0:BWIDTH-1] b1_out;
input  [0:BWIDTH-1] b1_in;
always @ (b1_in)
  begin : shift
    b1_out[0]=b1_in[2];
    b1_out[1]=b1_in[3];
    b1_out[2]=b1_in[4];
    b1_out[3]=b1_in[5];
    b1_out[4]=b1_in[6];
    b1_out[5]=b1_in[7];
    b1_out[6]=b1_in[8];
    b1_out[7]=b1_in[9];
```

```
        b1_out[8]=b1_in[10];
        b1_out[9]=b1_in[11];
        b1_out[10]=b1_in[12];
        b1_out[11]=b1_in[13];
        b1_out[12]=b1_in[14];
        b1_out[13]=b1_in[15];
        b1_out[14]=b1_in[16];
        b1_out[15]=b1_in[17];
        b1_out[16]=b1_in[18];
        b1_out[17]=b1_in[19];
        b1_out[18]=b1_in[20];
        b1_out[19]=b1_in[21];
        b1_out[20]=b1_in[22];
        b1_out[21]=b1_in[23];
        b1_out[22]=b1_in[24];
        b1_out[23]=b1_in[25];
        b1_out[24]=b1_in[26];
        b1_out[25]=b1_in[27];
        b1_out[26]=b1_in[28];
        b1_out[27]=b1_in[29];
        b1_out[28]=b1_in[30];
        b1_out[29]=b1_in[31];
        b1_out[30]=b1_in[0];
        b1_out[31]=b1_in[1];
    end
endmodule
```

本程序就是通过给位赋值来达到循环移位目的的。

4.2.2　S 盒设计

与其他对称密码体制类似，在轮函数加密过程中的字节代换也是通过 S 盒来完成的，字节与字节在 S 盒代换中是一一对应的，可以通过表格体现出来，如表 4.1 所示。

表 4.1　SM4 算法的 S 盒数据

行＼列	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	D6	90	E9	FC	CC	E1	3D	B7	16	B6	14	C2	28	FB	2C	05
1	2B	67	9A	76	2A	BE	04	C3	AA	44	13	26	49	86	06	99
2	9C	42	50	F4	91	EF	98	7A	33	54	0B	43	ED	CF	AC	62
3	E4	B3	1C	A9	C9	08	E8	95	80	DF	94	FA	75	8F	3F	A6
4	47	07	A7	FC	F3	73	17	BA	83	59	3C	19	E6	85	4F	A8
5	68	6B	81	B2	71	64	DA	B8	F8	EB	0F	4B	70	56	9D	35
6	1E	24	0E	5E	63	58	D1	A2	25	22	7C	3B	01	21	78	87
7	D4	00	46	57	9F	D3	27	52	4C	36	02	E7	A0	C4	C8	9E

续表

行＼列	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
8	EA	BF	8A	D2	40	C7	38	B5	A3	F7	F2	CE	F9	61	15	A1
9	E0	AE	5D	A4	9B	34	1A	55	AD	93	32	30	F5	8C	B1	E3
A	1D	F6	E2	2E	82	66	CA	60	C0	29	23	AB	0D	53	4E	6F
B	D5	DB	37	45	DE	FD	8E	2F	03	FF	6A	72	6D	6C	5B	51
C	8D	1B	AF	92	BB	DD	BC	7F	11	D9	5C	41	1F	10	5A	D8
D	0A	C1	31	88	A5	CD	7B	BD	2D	74	D0	12	B8	E5	B4	B0
E	89	69	97	4A	0C	96	77	7E	65	B9	F1	09	C5	6E	C6	84
F	18	F0	7D	EC	3A	DC	4D	20	79	EE	5F	3E	D7	CB	39	48

S 盒的设计主要是建立一个存储表格信息的对应程序：

```
always @(sin)
 case({sin[4:7],sin[0:3]})
   8'h00:  sout= 8'hd6;
   8'h10:  sout= 8'h90;
   8'h20:  sout= 8'he9;
   8'h30:  sout= 8'hfe;
   8'h40:  sout= 8'hcc;
   ......
   8'hcf:  sout= 8'hd7;
   8'hdf:  sout= 8'hcb;
   8'hef:  sout= 8'h39;
   8'hff:  sout= 8'h48;
  default:  sout= 8'h00;
 endcase
```

通过上述程序来完成表 4.1 中的 S 盒字节代换信息。例如，加密时字节 0x00 用 0xd6 来替代，可以用下列程序表达出：

```
case({sin[4:7],sin[0:3]})
8'h00: sout= 8'hd6
```

4.2.3 密钥扩展设计

密钥扩展算法：SM4 算法中加密算法的轮密钥由加密密钥通过密钥扩展算法生成。

设加密密钥 $\mathrm{MK}=(\mathrm{MK}_0,\mathrm{MK}_1,\mathrm{MK}_2,\mathrm{MK}_3)$，$\mathrm{MK}_i\in Z_2^{32}$，$i=0,1,2,3$；令轮密钥为 $K_i\in Z_2^{32}$，$i=0,1,\cdots,31$，则轮密钥生成方法如下。

首先，$(K_0,K_1,K_2,K_3)=(\mathrm{FK}_0\oplus\mathrm{MK}_0,\mathrm{FK}_1\oplus\mathrm{MK}_1,\mathrm{FK}_2\oplus\mathrm{MK}_2,\mathrm{FK}_3\oplus\mathrm{MK}_3)$

然后，对 $i=0,1,2,\cdots,31$

$$\mathrm{rk}_i=K_{i+4}=K_i\oplus T'(K_{i+1}\oplus K_{i+2}\oplus K_{i+3}\oplus \mathrm{CK}_i)$$

其中：

（1）T' 变换与加密算法轮函数中的 T 基本相同，只要将其中的线性变换 L 修改为以下 L'：$L'(B)=B\oplus(B<<<13)\oplus(B<<<23)$；

（2）系统参数 $\mathrm{FK}=(\mathrm{FK}_0,\mathrm{FK}_1,\mathrm{FK}_2,\mathrm{FK}_3)$，其取值采用 16 进制表示为：

FK_0=A3B1BAC6，FK_1=56AA3350，FK_2=677D9197，FK_3=B27022DC。

（3）固定参数 CK_i（$i=0,1,\cdots,31$），其选取方法为：设 $\mathrm{ck}_{i,j}$ 为 CK_i 的第 j 字节（$i=0,1,\cdots,31$；$j=0,1,2,3$），即 $\mathrm{CK}_i=(\mathrm{ck}_{i,0},\mathrm{ck}_{i,1},\mathrm{ck}_{i,2},\mathrm{ck}_{i,3})\in(Z_2^8)^4$，则取 $\mathrm{ck}_{i,j}=(4i+j)\times 7(\mathrm{mod}256)$。32 个固定参数 CK_i 的 16 进制表示如表 4.2 所示。

表 4.2　CK_i 数据表

00070e15	1c232a31	383f464d	545b6269
70777e85	8c939aa1	a8afb6bd	c4cbd2d9
e0e7eef5	fc030a11	181f262d	343b4249
50575e65	6c737a81	888f969d	a4abb2b9
c0c7ced5	dce3eaf1	f8ff060d	141b2229
30373e45	4c535a61	686f767d	848b9299
a0a7aeb5	bcc3cad1	d8dfe6ed	f4fb0209
10171e25	2c333a41	484f565d	646b7279

如图 4.3 所示，密钥扩展流程清晰地反映出密钥扩展的整个过程，其程序实现如程序 4-2 所示。

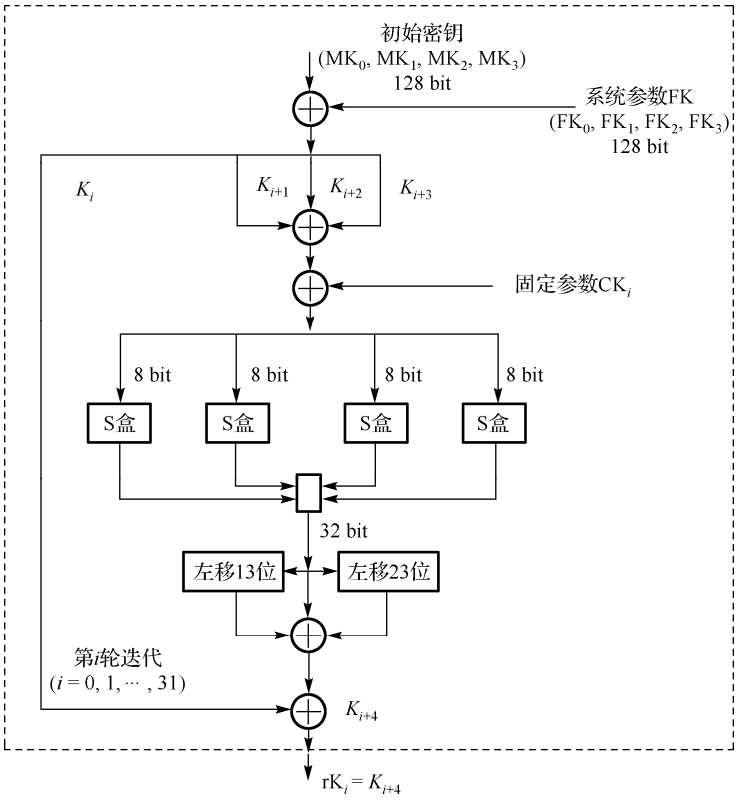

图 4.3　SM4 算法的密钥扩展流程

程序 4-2：

```
always @ (posedge clk or negedge reset)
  if (!reset)
    key_reg<=0;
      else
        key_reg<=next_key;
always @ (get_key or exp_run or key_in or key_reg or next_3 or next_2 or
          next_1 or next_0)
  begin
   if(get_key)
next_key={key_in[0:31]^FK0,key_in[32:63]^FK1,key_in[64:95]^FK2,key_in[
          96:127]^FK3};
   else
    if(exp_run)
        next_key={next_0,next_1,next_2,next_3};
          else
          next_key=key_reg;
  end
always @ (mode or cur_3 or cur_2 or cur_1 or cur_0 or b_shift2 or b_shift1
          or b_shift0)
  begin
    if(mode)
     begin
       next_0=cur_1;
       next_1=cur_2;
       next_2=cur_3;
       next_3=cur_0^b_shift2^b_shift1^b_shift0;
     end
       else
           begin
             next_3=cur_2;
             next_2=cur_1;
             next_1=cur_0;
             next_0=cur_3^b_shift2^b_shift1^b_shift0;
           end
   end
always @ (cur_3 or cur_2 or cur_1 or cur_0 or mode or ck)
begin
    case (mode)
      1'b1 : fir_ad=ck^cur_3^cur_2^cur_1;
      1'b0 : fir_ad=ck^cur_2^cur_1^cur_0;
     default : fir_ad=0;
    endcase
```

```
end
always @(key_reg)
begin
  cur_3 = key_reg[96:127];
  cur_2 = key_reg[64:95];
  cur_1 = key_reg[32:63];
  cur_0 = key_reg[0:31];
  key_out = key_reg;
  round_key=key_reg[96:127];
end
CK    Ck(
        .ck(ck),
        .counter(counter)
        );
sbox  Sbox3(
          .sout(sbox3),
          .sin(fir_ad[24:31])
          );
sbox  Sbox2(
          .sout(sbox2),
          .sin(fir_ad[16:23])
          );
sbox  Sbox1(
          .sout(sbox1),
          .sin(fir_ad[8:15])
          );
sbox  Sbox0(
          .sout(sbox0),
          .sin(fir_ad[0:7])
          );
BK_SHIFT2  B_shift2(
                 .b2_out(b_shift2),
                 .b2_in({sbox0,sbox1,sbox2,sbox3})
                 );
BK_SHIFT1  B_shift1(
                 .b1_out(b_shift1),
                 .b1_in({sbox0,sbox1,sbox2,sbox3})
                 );
BK_SHIFT0  B_shift0(
                 .b0_out(b_shift0),
                 .b0_in({sbox0,sbox1,sbox2,sbox3})
                 );
```

密钥扩展函数与轮函数加密的设计类似，只是密钥扩展函数的输入是初始密钥和系统

参数相与的结果，其中包括字节代换、循环移位、不同字节的整合等操作，来达到密钥扩展的目的。

4.2.4 轮函数加密设计

轮函数加密步骤如下。

假定明文输入为：$(X_0,X_1,X_2,X_3)\in(Z_2^{32})^4$，密文输出为：$(Y_0,Y_1,Y_2,Y_3)\in(Z_2^{32})^4$，子密钥为：$\mathrm{rk}_0,\mathrm{rk}_1,\cdots,\mathrm{rk}_{31}\in Z_2^{32}$，则

$$X_{i+4}=F(X_i,X_{i+1},X_{i+2},X_{i+3},rk_i)=X_i\oplus T(X_{i+1}\oplus X_{i+2}\oplus X_{i+3}\oplus rk_i),\quad i=0,1,\cdots,31$$

$$(Y_0,Y_1,Y_2,Y_3)=(X_{35},X_{34},X_{33},X_{32})$$

其中，以 32 位为单位的逆序输出是为了加解密的一致性。解密算法与加密算法的结构类似，只是轮密钥的使用顺序相反。加密算法轮密钥的使用顺序为：$\mathrm{rk}_0,\mathrm{rk}_1,\cdots,\mathrm{rk}_{31}$。

合成置换 T：

$Z_2^{32}\to Z_2^{32}$ 是一个可逆变换，由非线性变换 τ 和线性变换 L 复合而成，即 $T(.)=L(\tau(.))$。

非线性变换 τ：

τ 由 4 个并行的 S 盒构成，S 盒为固定的 8bit 输入 8bit 输出的置换，记为 Sbox(·)。

设输入为 $A=(a_0,a_1,a_2,a_3)\in(Z_2^8)^4$，输出为 $B=(b_0,b_1,b_2,b_3)\in(Z_2^8)^4$，则 $(b_0,b_1,b_2,b_3)=\tau(A)=(\mathrm{Sbox}(a_0),\mathrm{Sbox}(a_1),\mathrm{Sbox}(a_2),\mathrm{Sbox}(a_3))$

线性变换 L：

非线性变换 τ 的输出是线性变换 L 的输入。设输入为 $B\in Z_2^{32}$，输出为 $C\in Z_2^{32}$，则 $C=L(B)=B\oplus(B<<<2)\oplus(B<<<10)\oplus(B<<<18)\oplus(B<<<24)$。

程序 4-3：

```
always @ (cur_3 or cur_2 or cur_1 or cur_0 or b_shift4 or b_shift3 or b_shift2
          or b_shift1 or b_shift0)
  begin
    next_0=cur_1;
    next_1=cur_2;
    next_2=cur_3;
    next_3=cur_0^b_shift4^b_shift3^b_shift2^b_shift1^b_shift0;
  end
always @ (cur_3 or cur_2 or cur_1 or round_key)
 begin
  fir_ad=cur_3^cur_2^cur_1^round_key;
 end
always @ (sbox3 or sbox2 or sbox1 or sbox0)
   sbox_out={sbox0,sbox1,sbox2,sbox3};
sbox   Sbox3(
            .sout(sbox3),
            .sin(fir_ad[24:31])
            );
```

```
    sbox   Sbox2(
                .sout(sbox2),
                .sin(fir_ad[16:23])
                );
    sbox   Sbox1(
                .sout(sbox1),
                .sin(fir_ad[8:15])
                );
    sbox   Sbox0(
                .sout(sbox0),
                .sin(fir_ad[0:7])
                );
    B_SHIFT4  B_shift4(
                      .b4_out(b_shift4),
                      .b4_in(sbox_out)
                      );
    B_SHIFT3  B_shift3(
                      .b3_out(b_shift3),
                      .b3_in(sbox_out)
                      );
    B_SHIFT2  B_shift2(
                      .b2_out(b_shift2),
                      .b2_in(sbox_out)
                      );
    B_SHIFT1  B_shift1(
                      .b1_out(b_shift1),
                      .b1_in(sbox_out)
                      );
    B_SHIFT0  B_shift0(
                      .b0_out(b_shift0),
                      .b0_in(sbox_out)
                      );
    always @(data_reg)
    begin
      cur_0 = data_reg[0:31];
      cur_1 = data_reg[32:63];
      cur_2 = data_reg[64:95];
      cur_3 = data_reg[96:127];
      data_out={data_reg[96:127],data_reg[64:95],data_reg[32:63],data_reg[0:31]};
    end
```

上述程序中，轮函数加密的设计主要完成一轮加密过程，主程序加密时可以循环调用轮函数加密模块来完成加密过程，其中包括字节代换、循环移位、不同字节的整合等操作。

4.3 SM4 算法工程实现

1．创建 ISE 工程

打开 ISE 开发环境，选择菜单 File→New Project，建立一个新工程，然后设置顶层文件模块名、存储目录和设计方式。接着选择器件，这里选择 Xilinx 的 Virtex4 来实现，最后再选择第三方仿真软件 ModelSim 和 Verilog 语言进行仿真测试。新工程的建立如图 4.4 和图 4.5 所示。

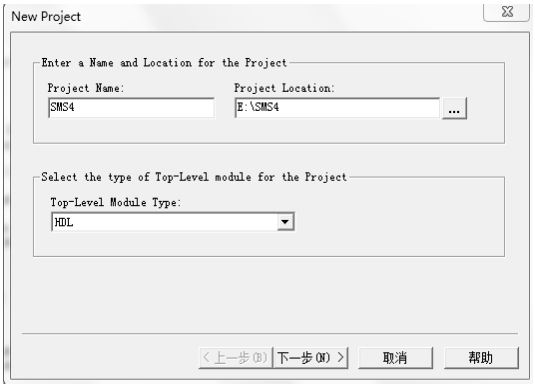

图 4.4　建立 SM4 工程文件

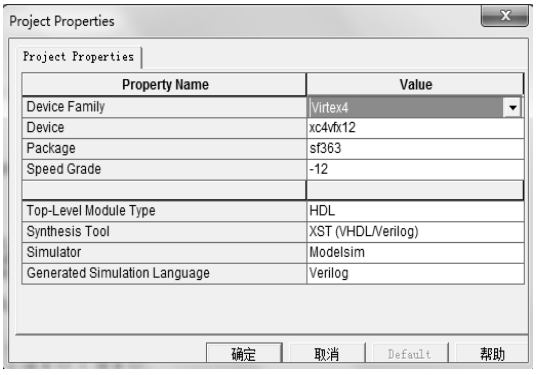

图 4.5　建立 SM4 工程文件属性设置图

2．编写设计代码

将 4.2 节中编写的模块代码导入新建的工程中，具体实现如图 4.6 所示。

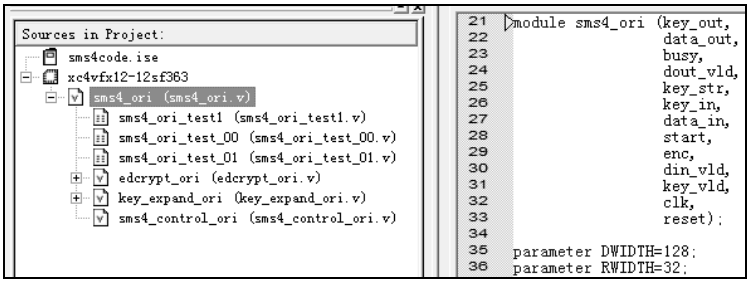

图 4.6　SM4 算法实现

3．工程验证

（1）根据以上编写的程序，编写该程序的测试文件，如图 4.7 所示。

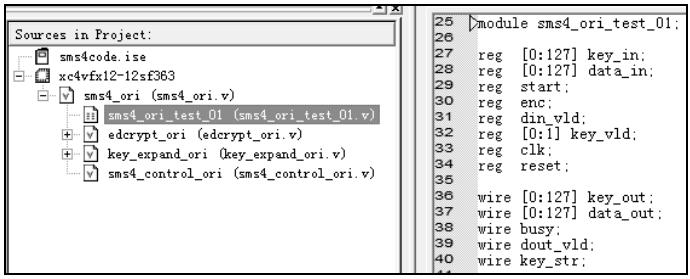

图 4.7　SM4 算法测试文件

（2）根据上一步的测试文件，利用 ModelSim 仿真软件，对该算法进行仿真测试，仿真结果如图 4.8～图 4.10 所示。

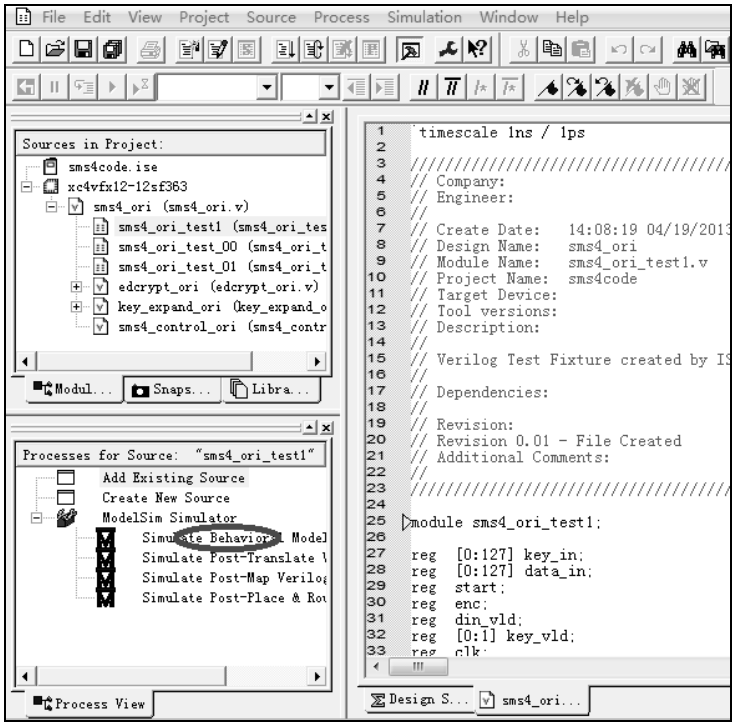

图 4.8　SM4 算法的 ModelSim 调用示意图

图 4.9　SM4 算法加密仿真结果

图 4.10　SM4 算法解密仿真结果

4.4 效果测试

通过功能仿真与系统测试，可以得到整个 SM4 算法 FPGA 设计的正确性，为了测试其性能，还需要对整个设计在芯片上的综合结果进行分析。

仿真时，在波形文件中需要给出合适的时钟周期。在 FPGA 程序设计中，验证算法的正确性和设计合理性的方法之一是对程序进行波形仿真，在本书中使用 ModelSim 对算法进行仿真，以验证其功能是否正确。验证时使用的明文（16 进制）为 8052310958432095739874658678598 5，密钥为 0123456789abcdeffedcba9876543210，加密结果为（16 进制）404cd32180dfa2c34f77b017c3f2b4d8。设置加解密标志位 enc 为 1，可以进行解密操作，将上述加密结果作为输入，密钥不变，可以得到明文为：80523109584320957398746586785985。

下面对该工程的时序、运行效率、资源占用情况进行分析。在 Xilinx 公司的 4vfx12sf363-12 元器件上，使用 ISE 的 XST 分析工具进行综合。综合结果如图 4.11 所示。

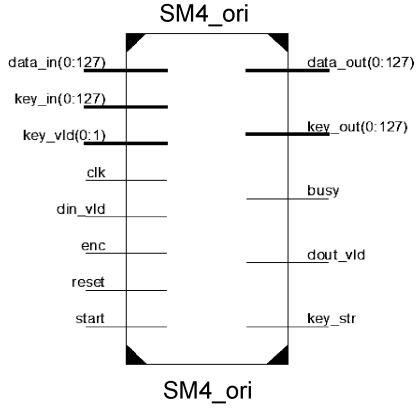

图 4.11　SM4 算法综合后的外部器件

资源占用情况分析报告如表 4.3 所示。

表 4.3　SM4 算法资源占用情况

Selected Device : 4vfx12sf363-12	
Number of Slices	1142 out of 5472 20%
Number of Slice Flip Flops	271 out of 10944 2%
Number of 4 input LUTs	2074 out of 10944 18%
IOB Flip Flops	1
Number of GCLKs	1 out of 32 3%

时序分析报告如下。

Timing constraint: Default period analysis for Clock 'clk'

Clock period: 6.299ns (frequency: 158.761MHz)

Total number of paths / destination ports: 282770 / 278

由时序分析报告结果可知，SM4 算法的最高工作频率为 158.761 MHz。分析程序可知，每 128 比特的分组数据，加密共用 68 个时钟。所以，SM4 算法的最高处理速度计算结果为 128×158.761/68=298.844 Mbps。

4.5 本章小节

分组加密算法 SM4 是我国于 2006 年 1 月公布的商用密码算法，该算法的分组长度为 128bit，密钥长度为 128bit。加密算法与密钥扩展算法都采用 32 轮非线性迭代结构。解密算法与加密算法的结构类似，只是轮密钥的使用顺序相反，解密轮密钥是加密轮密钥的逆序。

本章首先对 SM4 算法进行了介绍，包括算法定义、轮函数、加/解密运算和密钥扩展算法等部分。接着对 SM4 算法的 FPGA 实现进行了描述，并给出了核心部分的代码和仿真结果。最后分别对 SM4 分组密码算法的 FPGA 设计方法和硬件实现性能进行了总结，使读者对 SM4 分组密码算法的 FPGA 实现有了更深入的认识。

第 5 章　RSA 算法 FPGA 实现

RSA 公钥加密算法是 1977 年由 Ron Rivest、Adi Shamirh 和 Len Adleman 在美国麻省理工学院开发的，RSA 的取名来自开发者的名字。

RSA 体制的安全性完全依赖于大整数分解问题，一般来说，只要其钥匙的长度足够长，用 RSA 加密的信息是不能被破解的。迄今为止，对 RSA 体制的攻击虽然已提出很多方法，但以目前的计算能力，1024 位或 2048 位的 RSA 加密算法还是相对安全的。

5.1　RSA 算法原理

5.1.1　参数产生与密钥生成

（1）选取两个大素数 p 和 q（目前两个数的长度都接近 512bit，是安全的），并计算两者的乘积得 $n = p \times q$。

（2）根据欧拉函数，不大于 n 且与 n互质的整数个数为 $\varphi(n) = (p-1)(q-1)$。

（3）随机选择整数 $e(1 < e < \varphi(n))$ 作为公钥，要求满足 $\gcd(e, \varphi(n)) = 1$，即 e与$\varphi(n)$ 互素。

（4）用欧几里德扩展算法计算私钥 d，以满足 $d \times e \equiv 1 \bmod(\varphi(n))$，即 $d = e^{-1} \bmod(\varphi(n))$。则 e 和 n 是公钥，d 是私钥。

注意：e 和 n 应公开，两个素数 p 和 q 不再需要，可销毁，但绝不能泄露。

参数选择的一些原则：除了需要选取足够大的大整数 n 外，对素数 p 和 q 的选取应该满足以下要求。

（1）为避免椭圆曲线被用因子分解法分解，p 和 q 的长度相差不能太大。如使用 1024bit 的模数 n，则 p 和 q 的模长都大致在 512bit。

（2）p 和 q 差值不应该太小。如果 p–q 太小，则 $p \approx q$，因此 $p \approx \sqrt{n}$，故 n 可以简单地用所有接近 $\sqrt{n}$ 的奇整数试除而被有效分解。

（3）$\gcd(p-1, q-1)$ 应该尽可能小。

（4）p 和 q 应为强素数，即 p–1 和 q–1 都应有大的素因子。

另外，为了防止低指数攻击，e 不能选取太小的数。由于解密和验证所花费的时间与解密算法中作为指数的私钥 d 的大小成正比，所以一些低速的设备上可能选择较小的私钥 d，但是这就给攻击者机会以攻破 RSA，当 $d < n^{1/4}$ 时，便有办法攻击 RSA，所以 d 也不应该太小。

5.1.2　加解密过程

加密过程如下。

首先将明文比特串分组，使得每个分组对应的十进制数小于 n，即分组长度小于 $\log_2 n$，然后对每个明文分组 m_i 作加密运算，具体过程分为如下几步：

第一步，获得接收方公钥 (e,n)；

第二步，把消息 M 分组为长度为 $L(L<\log_2 n)$ 的消息分组 $M=m_1m_2\cdots m_t$；

第三步，使用加密算法 $c_i=m_i^e \bmod n(1\leqslant i\leqslant t)$，计算出密文 $C=c_1c_2\cdots c_t$；

第四步，将密文 C 发送给接收方。

解密过程如下：

第一步，接收方收到密文 C 并把密文 C 按长度为 L 分组得 $C=c_1c_2\cdots c_t$；

第二步，使用私钥 d 和解密算法 $m_i=c_i^d \bmod n(1\leqslant i\leqslant t)$，计算 m_i；

第三步，得到明文消息 $M=m_1m_2\cdots m_t$。

5.1.3　正确性证明与安全性分析

（1）加解密正确性证明

由加密算法 $c_i=m_i^e \bmod n$，可得解密算法：

$$m_i\equiv c_i^d \bmod n\equiv m_i^{ed} \bmod n\equiv m_i^{k\varphi(n)+1} \bmod n$$

下面分两种情况进行讨论。

第一种情况：$\gcd(m_i,n)=1$，则由欧拉定理得

$$m_i^{\varphi(n)}\equiv 1 \bmod n,\ \ m_i^{k\varphi(n)}\equiv 1 \bmod n,\ \ m_i^{k\varphi(n)+1}\equiv m_i \bmod n$$

又因为 $m_i<n$，所以，$c_i^d \bmod n\equiv m_i^{k\varphi(n)+1}\equiv m_i \bmod n\equiv m_i$。

第二种情况：$\gcd(m_i,n)\neq 1$，可得 $\gcd(m_i,n)>1$，由于 $n=p\times q$，所以 $\gcd(m_i,n)$ 必含 p或q。设 $\gcd(n,m_i)=p$，则有 $m_i=tp,1\leqslant t\leqslant q$，由欧拉定理得

$$m_i^{\varphi(q)}\equiv 1 \bmod q$$

因此，对于任何 k，总有

$$m_i^{k\varphi(q)}\equiv 1 \bmod q,[m_i^{k\varphi(q)}]^{\varphi(q)}\equiv 1 \bmod q,\ \ m_i^{k\varphi(n)}\equiv 1 \bmod q$$

因此存在以整数 r，使得 $m_i^{k\varphi(n)}\equiv 1+rq$，两边同时乘以 $m_i=tp$得

$$m_i^{k\varphi(n)+1}\equiv m_i+rtpq=m_i+rtn$$

又因为 $m_i<n$，上面等式两边同时进行模 n 运算，得

$$m_i^{k\varphi(n)+1} \bmod n\equiv (m_i+rtn) \bmod n=m_i$$

得

$$m_i^{k\varphi(n)+1} \bmod n\equiv m_i$$

故，由上述过程验证了加密解密算法的正确性。

（2）安全性分析

一般说来，上述类型的大整数 n 的因子分解问题是困难的。其实，RSA 体制的问世大大刺激了大整数因子分解算法的研究。人们利用很多数学知识，结合计算机的应用，提出了不少有关大整数因子分解的算法。

在一个 RSA 体制中，$\varphi(n)$ 显然是不能泄露的，否则攻击者由公开的 e 应用辗转相除法很容易就能求出保密的 d。其实，知道 $\varphi(n)$ 也就可以分解 n：

$$p+q=n-\varphi(n)+1,\quad p-q=\sqrt{(p+q)^2-4n}$$

因此，攻击者设法找出 $\varphi(n)$ 并不会比直接分解 n 容易。另外，人们已经证明：如果有算法能求出解密指数 d，那么据此也就能以至少 1/2 的概率来分解 n，因此，一旦解密指数 d 泄露，模数 n 必须重新选择。

5.2 RSA 算法相关模块 FPGA 设计

首先，通过对算法结构的分析发现，实现该算法主要需要三个模块：第一个模块是 RSA 顶层模块，第二个模块是模幂状态控制模块，第三个模块是模乘算法的实现模块。RSA 算法 FPGA 实现的结构示意图如图 5.1 所示。

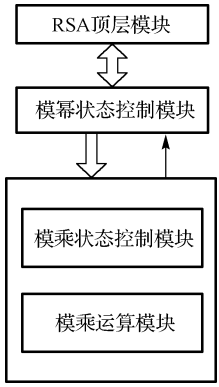

图 5.1 RSA 算法 FPGA 实现的关键模块

5.2.1 Montgomery 算法模块设计

Montgomery 模乘能将 RSA 中的模 n 运算转换为模 2^k，而模 2^k 能够通过取数的低 k 位来实现，因而避免了繁琐的模 n 除法运算。

下面简单介绍一下 Montgomery 算法的基本思想。

计算 $A*B*r^{-1}(\mathrm{mod}\,N)$；设 N 为 k 比特的整数，即：$2^{k-1}<N<2^k, A<N, B<N$；

$\gcd(N,r)=1$，N 和 r 互素，取 $r=2^k$；r^{-1} 是 r 模 N 的逆元，即：$r*r^{-1}=1\mathrm{mod}\,N$。

计算 $\mathrm{Mon}P(A,B,r,N)=A*B*r^{-1}(\mathrm{mod}\,N)$ 的算法如下。

$\mathrm{Mon}P(A,B,r,N)$

{

$t=A*B$;

$u=(t+(t*N'\bmod r)N)/r$;

if $u\geq N$

return $u-N$;

else return u

}

因为 $r=2^k$，所以对 r 求模以及对 r 整除的运算都可以通过简单的移位操作来完成，消除了复杂的传统除法运算。因为 Montgomery 算法计算的是 $A*B*r^{-1}(\mathrm{mod}\,N)$ 的值，并不是 $A*B(\mathrm{mod}\,N)$ 的值，所以还需要进行预计算或者后处理以消除 r^{-1} 的影响，其程序实现如程序 5-1 所示。

程序 5-1：

```
entity modmult is
    Generic (MPWID: integer := 32);
    Port ( mpand : in std_logic_vector(MPWID-1 downto 0);
           mplier : in std_logic_vector(MPWID-1 downto 0);
           modulus : in std_logic_vector(MPWID-1 downto 0);
           product : out std_logic_vector(MPWID-1 downto 0);
           clk : in std_logic;
              ds : in std_logic;
              reset : in std_logic;
              ready : out std_logic);
end modmult;
……
begin
    -- 最终结果
    product <= prodreg4(MPWID-1 downto 0);
    with mpreg(0) select
        prodreg1 <= prodreg + mcreg when '1',
                        prodreg when others;
    -- 减去模值和减去 2 倍的模值
    prodreg2 <= prodreg1 - modreg1;
    prodreg3 <= prodreg1 - modreg2;
    -- 当结果为负值的时候，表明减去的太多
    modstate <= prodreg3(mpwid+1) & prodreg2(mpwid+1);
    -- 选择正确的模块化结果并将其赋值
    with modstate select
        prodreg4 <= prodreg1 when "11",
                        prodreg2 when "10",
                        prodreg3 when others;
    -- 同时，从移位过后的被乘数中减去模数
    mcreg1 <= mcreg - modreg1;
    with mcreg1(MPWID) select
        mcreg2 <= mcreg when '1',
                      mcreg1 when others;
    ready <= first;
    combine: process (clk, first, ds, mpreg, reset) is
    begin
           if reset = '1' then
           first <= '1';
        elsif rising_edge(clk) then
```

```
                if first = '1' then
            -- 第一次，设置寄存器开始过程
    -- 输入值是样本值
                    if ds = '1' then
                        mpreg <= mplier;
                        mcreg <= "00" & mpand;
                        modreg1 <= "00" & modulus;
                        modreg2 <= '0' & modulus & '0';
                        prodreg <= (others => '0');
                        first <= '0';
                    end if;
                else
        -- 当被乘数的所有比特被移出，操作结束
                        if mpreg = 0 then
                        first <= '1';
                    else
                    -- 将被乘数左移 1 比特
                        mcreg <= mcreg2(MPWID downto 0) & '0';
                    -- 将乘数右移 1 比特
                        mpreg <= '0' & mpreg(MPWID-1 downto 1);
                    -- 复制中间结果
                        prodreg <= prodreg4;
                    end if;
                end if;
            end if;
        end process combine;
    end Behavioral;
```

程序中主要接口信号定义如下。

reset：电路复位标志；

mpand：乘数；

mplier：被乘数；

modulus：模数；

product：模乘运算的结果；

ds：使能端信号；

ready：算法执行结束标志，在算法结束后给出 ready 为 1，若算法未结束，ready 为 0。

在**程序 5-1** 中，定义一个整型变量 MPWID 作为此模块的宽度，当使用此模块时，修改此参数，可以达到控制宽度的目的。乘数和被乘数必须是一个小于模量的值。

5.2.2 R-L 模式模幂算法模块设计

R-L 模式二进制算法是指对幂指数的二进制位从右往左逐位进行扫描。算法描述如下。

输入： m,n,e ；

输出：$c = m^e (\mathrm{mod}\, n)$

{

$e = \sum_{i=1}^{n-1} e_i * 2^i$;

$c = 1$;

for$(i = 0; i \leqslant n-1; i++)$

{

$f(e_i = 1)$

$c = m * c(\mathrm{mod}\, n)$;

else

$m = m * m(\mathrm{mod}\, n)$

}

}

从上述的 R-L 模式算法描述中可以看出，两次模乘运算 $c = m * c(\mathrm{mod}\, n)$ 和 $m = m * m(\mathrm{mod}\, n)$，在当前轮运算中并没有顺序依赖关系，也就是说模乘运算和模方运算是相互独立、互不影响的，所以可以用两个模乘运算器并行工作。那么 R-L 模式算法就能够进行并行运算，这样可以提高运算的速度。但是要以增加一个模乘运算器为代价，即增加了硬件电路资源的占用。

RSA 算法中，在一次加密或者解密操作中，模数 N 和公钥 e 以及私钥 d 是确定的。算法的核心，即运算量最大的模块是模幂运算，该设计采用由 Montgomery 构造模乘算法和 R-L 扫描模幂算法来实现。这种方法只需要一个主要的模乘运算模块再加上一些其他的控制模块，一方面占用硬件资源少，另一方面经过优化之后，速度有一定的提升。

程序 5-2：

```
entity RSACypher is
   Generic (KEYSIZE: integer := 1024);
   Port (indata: in std_logic_vector(KEYSIZE-1 downto 0);
            inExp: in std_logic_vector(KEYSIZE-1 downto 0);
            inMod: in std_logic_vector(KEYSIZE-1 downto 0);
            cypher: out std_logic_vector(KEYSIZE-1 downto 0);
            clk: in std_logic;
            ds: in std_logic;
            reset: in std_logic;
            ready: out std_logic
            );
end RSACypher;
architecture Behavioral of RSACypher is
attribute keep: string;
component modmult is
   Generic (MPWID: integer);
   Port ( mpand : in std_logic_vector(MPWID-1 downto 0);
          mplier : in std_logic_vector(MPWID-1 downto 0);
```

```
            modulus : in std_logic_vector(MPWID-1 downto 0);
            product : out std_logic_vector(MPWID-1 downto 0);
            clk : in std_logic;
            ds : in std_logic;
               reset : in std_logic;
               ready: out std_logic);
    end component;
    signal modreg: std_logic_vector(KEYSIZE-1 downto 0);
                              --储存模值
    signal root: std_logic_vector(KEYSIZE-1 downto 0); -- 平方值
    signal square: std_logic_vector(KEYSIZE-1 downto 0);
    signal sqrin: std_logic_vector(KEYSIZE-1 downto 0);
    signal tempin: std_logic_vector(KEYSIZE-1 downto 0);
    signal tempout: std_logic_vector(KEYSIZE-1 downto 0);  -- 乘法的结果
    signal count: std_logic_vector(KEYSIZE-1 downto 0);
    signal multrdy, sqrrdy, bothrdy: std_logic;-- 完成乘法指示信号
    signal multgo, sqrgo: std_logic;    -- 触发信号相乘
    signal done: std_logic; -- 信号显示加密完成
    begin
        ready <= done;
        bothrdy <= multrdy and sqrrdy;
        -- 模块化的乘数来产生结果
        modmultiply: modmult
        Generic Map(MPWID => KEYSIZE)
        Port Map(mpand => tempin,
                    mplier => sqrin,
                    modulus => modreg,
                    product => tempout,
                    clk => clk,
                    ds => multgo,
                    reset => reset,
                    ready => multrdy);
        -- 利用模块化的乘法完成平方操作
        modsqr: modmult
        Generic Map(MPWID => KEYSIZE)
        Port Map(mpand => root,
                    mplier => root,
                    modulus => modreg,
                    product => square,
                    clk => clk,
                    ds => multgo,
                    reset => reset,
                    ready =>sqrrdy);
```

```
    -- 产生使能信号
    mngcount: process (clk, reset, done, ds, count, bothrdy) is
    begin
    -- 产生 count 和 done
                if reset = '1' then
            count <= (others => '0');
            done <= '1';
        elsif rising_edge(clk) then
            if done = '1' then
                if ds = '1' then
-- 第一次操作
                    count <= '0' & inExp(KEYSIZE-1 downto 1);
                    done <= '0';
                end if;
-- 第一次之后的操作
            elsif count = 0 then
                if bothrdy = '1' and multgo = '0' then
                    cypher <= tempout;      -- 设置输出值
                    done <= '1';
                end if;
            elsif bothrdy = '1' then
                if multgo = '0' then
                    count <= '0' & count(KEYSIZE-1 downto 1);
                end if;
            end if;
        end if;
    end process mngcount;
    -- 为平方设置输入值
    setupsqr: process (clk, reset, done, ds) is
    begin
            if reset = '1' then
            root <= (others => '0');
            modreg <= (others => '0');
        elsif rising_edge(clk) then
            if done = '1' then
                if ds = '1' then
        -- 第一次通过，只输入采样值
                    modreg <= inMod;
                    root <= indata;
                end if;
        -- 第一次后，把平方结果反馈给乘数
            else
                root <= square;
```

```
        end if;
    end if;
end process setupsqr;
-- 这个过程将为结果设置输入值
setupmult: process (clk, reset, done, ds) is
begin
        if reset = '1' then
        tempin <= (others => '0');
        sqrin <= (others => '0');
        modreg <= (others => '0');
    elsif rising_edge(clk) then
        if done = '1' then
            if ds = '1' then
    -- 如果指数的最低有效位是'1'，然后将消息值赋值给乘数
    -- 否则将乘数设置为 1
    -- 平方也设置成 1，所以第一次乘法的结果将是 1 或者消息值
                if inExp(0) = '1' then
                    tempin <= indata;
                else
                    tempin(KEYSIZE-1 downto 1) <= (others => '0');
                    tempin(0) <= '1';
                end if;
                modreg <= inMod;
                sqrin(KEYSIZE-1 downto 1) <= (others => '0');
                sqrin(0) <= '1';
            end if;
-- 第一轮后，乘法和平方的结果将反馈给乘数
--  counter 将向右移动 1 比特
-- 如果指数的最低有效位是'1'，将最近的平方操作值反馈给乘数
-- 否则,平方值设置为 1,表明没有乘法
        else
            tempin <= tempout;
            if count(0) = '1' then
                sqrin <= square;
            else
                sqrin(KEYSIZE-1 downto 1) <= (others => '0');
                sqrin(0) <= '1';
            end if;
        end if;
    end if;
end process setupmult;
crypto: process (clk, reset, done, ds, count, bothrdy) is
begin
```

```
                if reset = '1' then
            multgo <= '0';
        elsif rising_edge(clk) then
            if done = '1' then
                if ds = '1' then
        -- 第一次通过自动触发获得第一个乘数循环
                    multgo <= '1';
                end if;
        -- 第一次之后，两个操作完成时触发
            elsif count /= 0 then
                if bothrdy = '1' then
                    multgo <= '1';
                end if;
            end if;
        -- 乘数已经开始时，禁用乘法器的输入
                if multgo = '1' then
                    multgo <= '0';
                end if;
        end if;
    end process crypto;
end Behavioral;
```

程序中主要接口信号定义如下。

reset：电路复位标志，高位有效；

ds：使能信号；

clk：时钟信号；

ready：算法执行结束标志，在算法结束后给出 ready 为 1，若算法未结束，ready 为 0；

indata：待加密的数据，通过 KEYSIZE 控制宽度；

Cypher：加密结果；

inExp：加解密指数；

inMod：模数。

在**程序 5-2** 中，实现模幂算法的同时，也完成了 RSA 加解密算法的过程。当实现加密过程时，指数就是 RSA 算法的公钥；当实现解密过程时，指数就是 RSA 算法的私钥。

5.3 RSA 算法工程实现

1．创建 ISE 工程

打开 ISE 开发环境，选择菜单 File→New Project，建立一个新工程，然后设置顶层文件模块名、存储目录和设计方式。接着选择器件，这里选择 Xilinx 的 Virtex6 来实现。最后再选择第三方仿真软件 ModelSim 和 VHDL 语言进行仿真测试。新工程的建立如图 5.2 和图 5.3 所示。

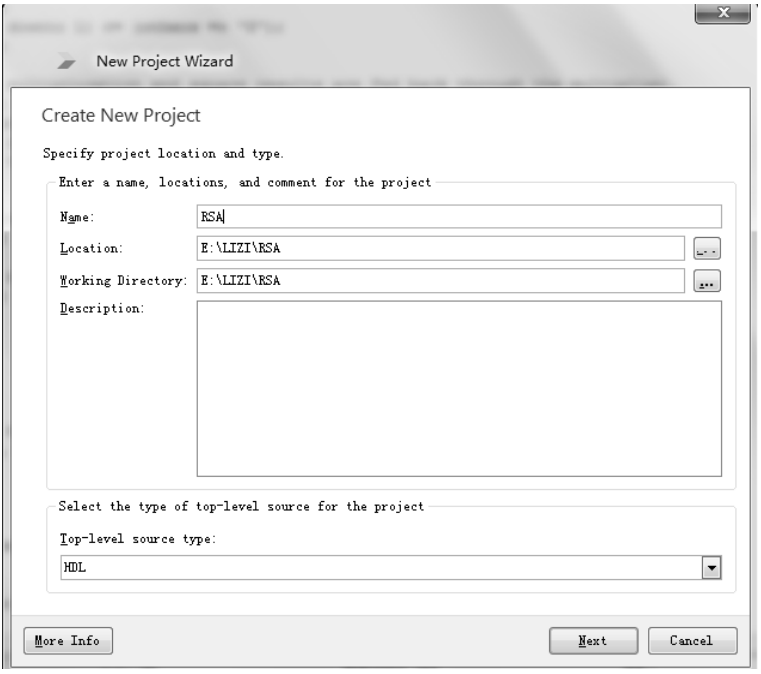

图 5.2　RSA 算法工程的建立

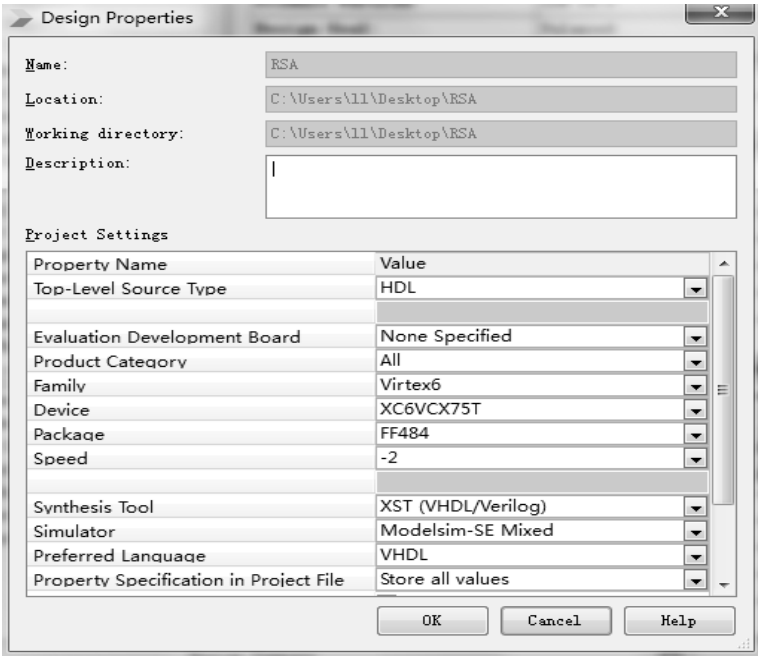

图 5.3　RSA 算法工程的设置

2．编写设计代码

在 5.2 节中详细叙述了 RSA 算法的关键模块设计，这里将上述模块在 ISE 中进行实现，模块具体实现如图 5.4 所示。

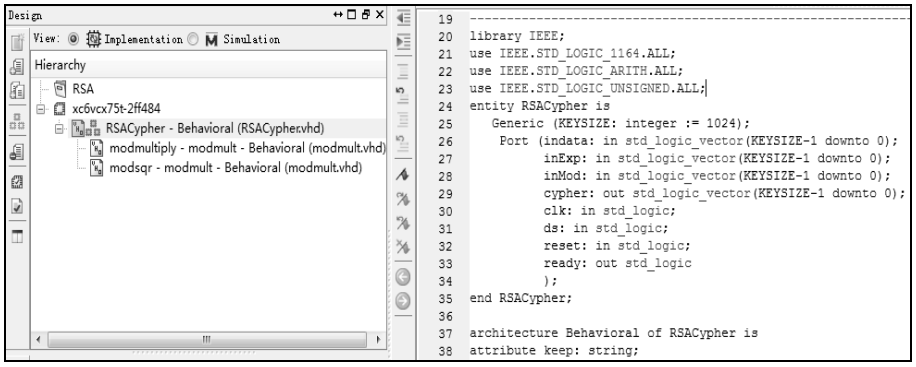

图 5.4　RSA 算法的实现

3．工程验证

（1）根据以上程序，编写该程序的测试文件，如图 5.5 和 5.6 所示。

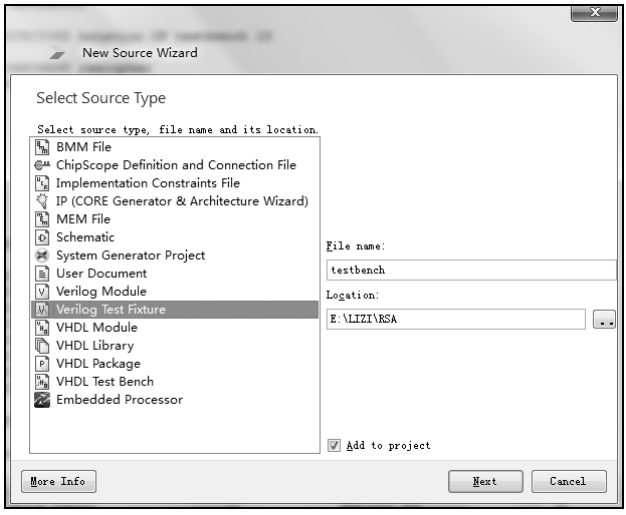

图 5.5　测试文件的建立

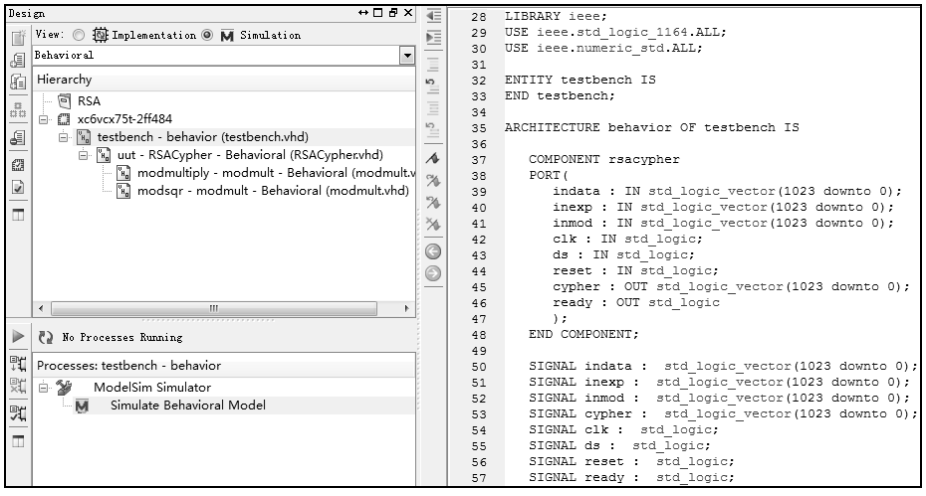

图 5.6　测试程序

（2）根据上一步的测试文件，调用 ModelSim 仿真软件，如图 5.7 所示，对该算法进行仿真测试，仿真结果如图 5.8 和图 5.9 所示。

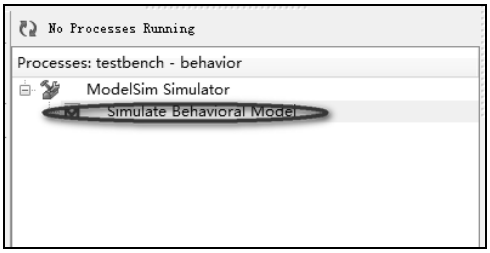

图 5.7　调用 ModelSim 仿真软件

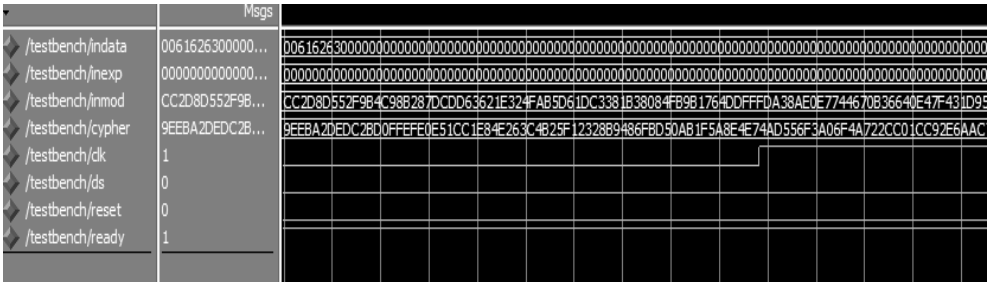

图 5.8　RSA 算法加密仿真图

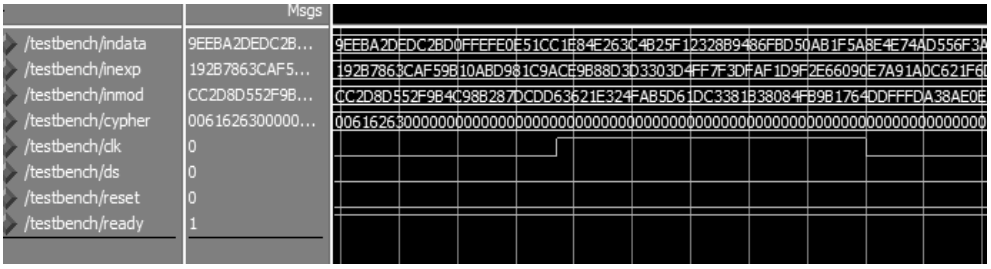

图 5.9　RSA 算法解密仿真图

5.4 效果测试

为了验证设计的正确性，本书采用 ISE 14.3 开发工具对系统设计进行了整合和综合，编写了 test bench；用 ModelSim SE10.1 仿真工具进行了仿真，包括各个子模块的仿真和整个系统的仿真；软件仿真通过后又将综合后的编程文件烧录到 FPGA 中进行验证。该测试所使用的计算机配置为：四核 i5-3470 CPU，4G 内存，32 位操作系统。

由于每个用户具有的 RSA 公私钥都不相同，所以本实例中的硬件部分并不包括公私钥对的生成部分，公私钥对的生成部分通过软件部分来实现：首先通过软件部分来生成公私钥密钥对，然后利用 RSA 算法 FPGA 实现加解密功能。

下面是 RSA 加解密部分的数据。

明文：00616263 00000000 00000000 00000000 00000000 00000000 00000000 00000000

00000000 00636261

加密密钥：00000000 00010001

解密密钥：192b7863 caf59b10 abd981c9 ace9b88d 3d3303d4 ff7f3dfa f1d9f2e6 6090e7a9 1a0c621f 6d88f0f2 0dbb8a3e 60cdd644 cfdb8245 53e46c00 5eb187ed 3551c48a b90c6583 421d5c9f 90ce27ff 9b76f384 8dd866ee 4ba4ce15 167fbd3f eda4cdce d4230ee7 7736c368 4aedf22e 311594b2c409d0a1 83b0d0f9 9837bfe6 fe9ede01

模值：cc2d8d55 2f9b4c98 b287dcdd 63621e32 4fab5d61 dc3381b3 8084fb9b 1764ddff fda38ae0 e7744670 b36640e4 7f431d95 a9812103 44d91dd7 b74e8231 a955c5c8 1a765dfb 33c03120 56a5c2a7 0e84a9f3 9f4e7297 eee5e561 58c8d14e 0168047e a3f19d89 71cc4818 b3805d54 6f2163ca b6916273 db6f3a3b a005750e 36b71151

密文：9eeba2de dc2bd0ff efe0e51cc1e84e26 3c4b25f1 2328b948 6fbd50ab 1f5a8e4e 74ad556f 3a06f4a7 22cc01cc 92e6aac7 f7bad4ee 19a49683 2f7027ca 822b7362 5a572b16 fcb5fe39 3347fef6 03bdf3ea a79b46bb 833a5f4c 393c6ddc 612c2504 b896a7b0 75ec821e 79d7bf1d 4d3c9cfc a41c416f 978c50be 9188c156 ae3af4e2

下面对该工程的时序、运行效率、资源占用情况进行分析。在 Xilinx 公司的 6vcx75tff484 FPGA 元器件上，运用 ISE 的 XST 分析工具进行综合。综合结果如图 5.10 所示。

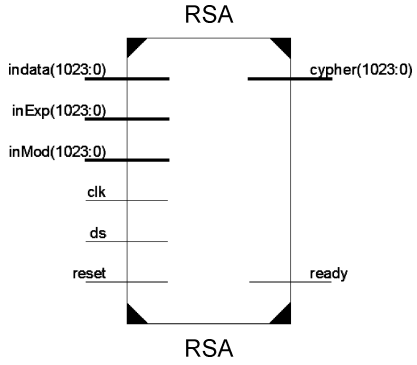

图 5.10　RSA 算法外部引脚

资源占用情况分析报告如表 5.1 所示。

表 5.1　RSA 算法资源占用情况

Selected Device : 6vcx75tff484-2				
Number of Slice Registers	14355	out of	93120	15%
Number of Slice LUTs	27299	out of	46560	58%
Number used as Logic	27299	out of	46560	58%
Number of BUFG/BUFGCTRLs	10253	out of	31401	32%
Number of BUFG/BUFGCTRLs	2	out of	32	6%

时序分析报告如下。

Speed Grade: -2

Minimum period: 20.771ns (Maximum Frequency: 48.143MHz)

Minimum input arrival time before clock: 1.826ns

Maximum output required time after clock: 1.124ns

Maximum combinational path delay: No path found

由时序分析报告结果可知，RSA 算法的最高工作频率为 48.143 MHz。分析程序可知，每个 1024 比特的分组需要经过 10000 个时钟。所以，RSA 算法的最高处理速度为 1024×48.143/10000≈5 Mbps。

由以上分析报告可以看出，整个工程在 6vcx75tff484 芯片上使用的查找表数为 27299 个，占芯片资源的 58%；使用寄存器芯片 14355 个，占寄存器资源的 15%。

5.5 本章小结

RSA 算法是一种非对称密码算法，加密密钥简称公钥，解密密钥简称私钥，公钥是公开的，私钥必须保密。它的安全性完全依赖于大整数分解问题，迄今为止，对 RSA 体制的攻击已提出很多方法，但以目前的计算能力，1024 或 2048 位的 RSA 加密算法还是相对安全的。因此，RSA 算法被广泛应用于身份认证、数字签名等信息交换领域。

本章首先对 RSA 算法的原理、工作模式与安全性进行了介绍，然后重点介绍了 RSA 算法的 FPGA 实现方法，包括模乘和模幂算法的实现，给出了硬件实现方案和相关代码，最后给出了仿真结果和综合结果，并对结果进行了分析，使读者不仅能从理论上了解 RSA 算法，更能从算法的 FPGA 硬件实现过程中加深对算法的理解。

第 6 章　ECC 算法 FPGA 实现

1985 年，N.Koblitz 和 V.Miller 分别独立提出了椭圆曲线密码体制（Elliptic Curve Cryptosystem，ECC），这是一种高安全性、高效率的公钥密码体系，它在密钥强度、加解密处理速度和存储开销上都有着明显的优势。采用椭圆曲线密码技术使密钥协商协议可以在较小密钥量下提供较高的安全性，不但所需带宽明显减少，而且还大大降低了用户端的计算负担和存储要求。

下面先简要介绍 ECC 密码体制的相关算法。

6.1　ECC 算法原理

6.1.1　参数产生

主要参数介绍及参数选取的原则。

定义在 F_{2^m} 上的椭圆曲线方程为：$y^2+xy=x^3+ax^2+b$（$a,b\in F_{2^m}$，且 $b\neq 0$）。椭圆曲线 $E(F_{2^m})$ 定义为：$E(F_{2^m})=\{(x,y)\,|\,y^2+xy=x^3+ax^2+b,x,y\in F_{2^m}\}\cup O$，其中 O 表示无穷远点。椭圆曲线 $E(F_{2^m})$ 上的点数用 $\#E(F_{2^m})$ 表示，称为椭圆曲线 $E(F_{2^m})$ 的阶。ECC 算法在 F_{2^m} 域上的椭圆曲线系统参数包括：

① 域的规模 $q=2^m$；

② 一个长度至少为 192 的比特串 SEED（选项）；

③ F_{2^m} 中的两个元素 a 和 b，它们用来定义椭圆曲线 E 的方程：$y^2+xy=x^3+ax^2+b$；

④ 基点 $G=(x_G,y_G)\in E(F_{2^m}),G\neq O$；

⑤ 基点 G 的阶 n；

⑥ 余因子 $h=\#E(F_{2^m})/n$（选项）。

密钥生成部分：选取一条阶为 kt、安全的椭圆曲线 $E(a,b)$，其中 k 为小整数，t 为大素数，在 $E(a,b)$ 上找一个阶为 t 的点 G。以用户 A 向用户 B 发送消息 m 为例，B 随机选取一个整数 $k(1<k<t)$ 作为私钥，计算：$P_{\mathrm{B}}=kG$ 作为用户 B 的公钥。

6.1.2　加密解密过程

用 ECC 算法进行加解密的过程如下。

① 用户 B 选定一条椭圆曲线 $E_p(a,b)$，并取椭圆曲线上一点，作为基点 G；

② 用户 B 选择一个私钥 k，并生成公钥 $P_{\mathrm{B}}=kG$；

③ 用户 B 将 $E_p(a,b)$ 和公钥 P_{B}、G 公开；

④ 用户 A 接到公开信息后，将待传输的明文编码到 $E_p(a,b)$ 上的一点 M（编码方法很多，这里不作讨论），并产生一个随机整数 r（其中 $r<n$）；

⑤ 用户 A 计算点 $C_1=M+rK$， $C_2=rG$；

⑥ 用户 A 将 C_1、C_2 传给用户 B；

⑦ 用户 B 接到信息后，计算 $C_1-kC_2=M+rK-krG=M+rK-rK=M$，结果就是点 M，即明文。

6.2 ECC 算法相关模块 FPGA 设计

通过对算法结构的分析发现，实现该算法主要需要以下几个模块：有限域乘法控制模块、有限域加法控制模块、有限域平方控制模块、有限域模逆控制模块、点加和倍加控制模块、点乘状态机控制模块。ECC 算法的结构示意图如图 6.1 所示。系统主要由有限域运算模块层和运算控制层电路构成。

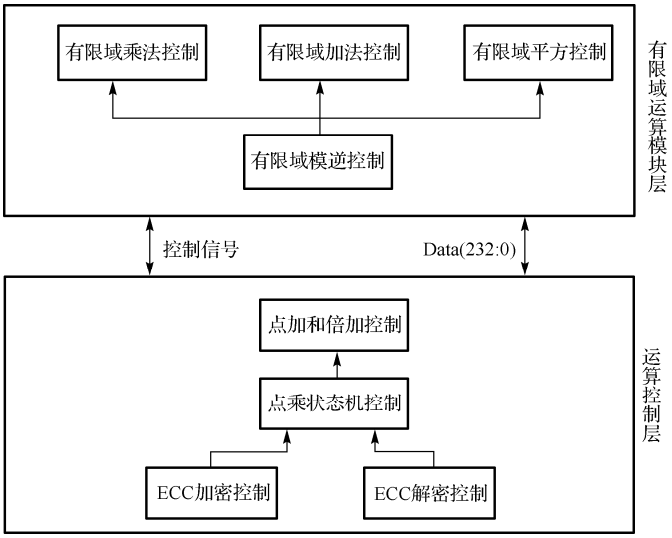

图 6.1 ECC 算法结构示意图

6.2.1 有限域加法的 FPGA 实现

有限域 F_{2^m} 中的加法运算就是将表示元素的二进制序列进行按位异或运算，因而可以使用简单的组合逻辑来实现，如程序 6-1 所示。

程序 6-1：

```
module MOD_ADD(
                DIN1,
                DIN2,
                DOUT
                );
input [232 : 0] DIN1;
```

```
input [232 : 0] DIN2;
output [232 : 0] DOUT;
assign   DOUT = DIN1 ^ DIN2;
endmodule
```

可以看出，只需要用异或门就可以实现。用 Veri1og HDL 语言进行描述，只需 assign $c = a\hat{}b$ 即可。

6.2.2　有限域乘法的 FPGA 实现

域乘法是整个设计中最关键的模块，它包括多项式相乘和取模两个过程。常见的乘法器设计思路是将两个 m 位操作数相乘，然后对 $f(x)$取模。这样就需要一个 $2m$ 的寄存器来存储中间结果，算法执行的效率不高。实现多项式模乘最基本的方法是移位相加法，具体算法如下。

输入：$A(x) = a_{n-1}x^{n-1} + \cdots + a_1 x + a_0$　　$B(x) = b_{n-1}x^{n-1} + \cdots + b_1 x + b_0$

输出：$C(x) = A(x) * B(x) \bmod f(x)$

（1）如果 $a_0 = 1$，则 $C = B$；

（2）i 从 1 到 $n-1$ 循环，$B = (B * X) \bmod f(x)$，如果 $a_i = 1$，则 $C = (C + B) \bmod f(x)$；

（3）返回 C。

其实现代码见程序 6-2。

程序 6-2：

```
always @(posedge CLK) begin
     if(!RST_N) begin
          data_in1[232:0] <= {233{1'b0}};
        data_in2[232:0] <= {233{1'b0}};
     end
     else if(IN_VALID) begin
          data_in1[232:0] <= DIN1[232:0];
     data_in2[232:0] <= DIN2[232:0];
   end
   else begin
      data_in1[232:0] <= {1'b0,{data_in1[232:1]}};
         data_in2[232:0] <= {{data_in2[231:0]},1'b0};
         if(data_in2[231] == 1'b1) begin
             data_in2[1]  <= data_in2[1] ^ 1'b1;
             data_in2[75] <= data_in2[75] ^ 1'b1;
         end
   end
end
always @(posedge CLK) begin
     if(IN_VALID) begin
     if(DIN1[0] == 1'b1)
        data_tmp <= DIN2;
```

```
        else
          data_tmp <= {233{1'b0}};
      end
      else begin
         if(data_in1[1] == 1'b1)
           data_tmp <= data_tmp ^ {{data_in2[231:0]},1'b0};
         else
           data_tmp <= data_tmp;
      end
    end
    always @(posedge CLK) begin
        if(!RST_N)
           cnt <= 8'h00;
      else if(IN_VALID)
        cnt <= 8'h00;
      else
        cnt <= cnt + 1;
    end
    always @(posedge CLK)  //增加一级缓存输出
     if(!RST_N) begin
        OUT_VALID<=0;
        OUT_VALID_TMP<=0;
        DOUT <= {233{1'b0}};
     end
     else if(cnt == 8'hE9) begin
          …
         end
     else begin
        OUT_VALID_TMP <= 1'b0;
        OUT_VALID<=1'b0;
        DOUT <= data_tmp;
     end
```

该算法需要进行大量的移位操作，不太适合基于处理器的实现方案，但是对于硬件实现来说是很方便的，而且不需要占用任何逻辑单元，所以在用 FPGA 进行硬件实现时，采用的是移位操作的方法。

6.2.3 有限域平方的 FPGA 实现

在硬件实现上，特定域上的算术部件与一般域上的算术部件，其复杂度和效率有很大的区别。模平方运算采用通用的算术部件，通常需要花费 $n/2$ 个时钟周期，若选用多项式基时，可以在一个时钟周期内完成，并且所用的资源最多为 n 个异或门，实现延时很小。下面将具体描述采用三项式基时平方运算的 FPGA 实现算法。

令 $A(x)=\sum_{i=0}^{n-1}a_i x^i$ 为GF(2^n) 上的任意元素，

$$q(x)=\sum_{i=0}^{n-1}q_i x^i = A^2(x) \bmod f(x) = a_0 + a_1 x + \cdots a_{n/2}x^n \cdots + a_{n-1}x^{2n-2} \bmod f(x)\ ,$$

令 $A(x^2)=\sum_{i=0}^{n-1}a_i x^{2i}=\sum_{i=0}^{2n-2}b_i x^i$ ，其中当 i 为偶数，即 $i=0,2,4,\cdots 2n-2$ 时， $b_i=a_{i/2}$ ；当 i 为奇数，即 $i=1,3,5,\cdots 2n-3$ 时， $b_i=0$ ；

令 $c_i=\sum_{i=0}^{n+2i}b_i x^i \bmod f(x)$ ，$i=0,1,\cdots,\frac{n}{2}-1$（当 n 为偶数时）；$i=0,1,\cdots,\frac{n-3}{2}$（当 n 为奇数时）；

则 $c_{\frac{n}{2}-1}=\sum_{i=0}^{2n-2}b_i x^i \bmod f(x)$ （当 n 为偶数时）；

$c_{\frac{n-1}{2}-1}=\sum_{i=0}^{2n-3}b_i x^i \bmod f(x)$ （当 n 为奇数时）；

所以 $q(x)=\sum_{i=0}^{2n-2}b_i x^i \bmod f(x)=c_{\frac{n}{2}-1}$ （n 为偶数时）；

$q(x)=\sum_{i=0}^{2n-2}b_i x^i \bmod f(x)=c_{\frac{n-1}{2}-1}+b_{2n-2}x^{2n-2} \bmod f(x)$ （当 n 奇数时）；

则 $c_0=\sum_{i=0}^{n}b_i x^i=\sum_{i=0}^{n-1}b_i x^i+b_n x^n=\sum_{i=0}^{n-1}b_i x^i+b_n+b_n x^k$ ；

如果当 n 为偶数时， $b_n=a_{\frac{n}{2}}$ ， 即 $c_0=\sum_{i=0}^{n-1}b_i x^i+b_n+b_n x^k$ ；

当 n 为奇数时， $b_n=0$ ， $c_0=\sum_{i=0}^{n-1}b_i x^i$ ；

$$A(x^2)=\sum_{i=0}^{2n-2}b_i x^i = c_0 + b_{n+1}x^{n+1} + \cdots + b_{2n-2}x^{2n-2}$$
$$= c_0 + b_{n+1}x^{n+1} + \cdots + b_{2n-2}x^{2n-2}$$

选用 $n=191, k=9$ ，则 $b_n, b_{n+2}, b_{n+4}, \cdots, b_{2n-1}, \cdots$ 为 0，

令 $c_1=\sum_{i=0}^{n+1}b_i x^i = c_0 + b_{n+1}x^{n+1} = \sum_{i=0}^{n-1}b_i x^i + b_{n+1}x + b_{n+1}x^{k+1}$;

$$c_2=\sum_{i=0}^{n+3}b_i x^i = c_1 + b_{n+3}x^{n+3} = \sum_{i=0}^{n-1}b_i x^i + b_{n+1}x + b_{n+1}x^{k+1} + b_{n+3}x^3 + b_{n+3}x^{k+3};$$

...

$$c_{\frac{n-1}{2}}=\sum_{i=0}^{2n-2}b_i x^i = c_{\frac{n-3}{2}} + b_{2n-2}x^{2n-2} = \sum_{i=0}^{n-1}b_i x^i + b_{n+1}x + b_{n+1}x^{k+1} + b_{n+3}x^3 + b_{n+3}x^{k+3} \ldots$$
$$+ b_{2n-2}x^{k-2} + b_{2n-2}x^{2k-2}$$

i 为偶数时

$$q_i = b_i (i < k+1);$$
$$q_i = b_i + b_{n-k+1} + b_{2n-2k+i} (k+1 < i < 2k-2);$$
$$q_i = b_i + b_{n-k+1} \text{（其他）}$$

i 为奇数时

$$q_i = b_{2n-k+i} (i < k);$$
$$q_i = b_{n-k+1} \text{（其他）}$$

ECC 算法平方运算的实现程序如程序 6-3 所示。

程序 6-3：

```
module   MOD_SQUA(DIN, DOUT);
input   [232:0]   DIN;
output  [232:0]   DOUT;
OUT_VALID;
/****************1*******************/
assign DOUT[0] = DIN[0] ^ DIN[196];
assign DOUT[2] = DIN[1] ^ DIN[197];
assign DOUT[4] = DIN[2] ^ DIN[198];
…
assign DOUT[72] = DIN[36] ^ DIN[232];
/*****************2******************/
assign DOUT[1] = DIN[117];
assign DOUT[3] = DIN[118];
assign DOUT[5] = DIN[119];
…
assign DOUT[73] = DIN[153];
/*****************3******************/
assign DOUT[74] = DIN[37] ^ DIN[196];
assign DOUT[76] = DIN[38] ^ DIN[197];
assign DOUT[78] = DIN[39] ^ DIN[198];
…
assign DOUT[146] = DIN[73] ^ DIN[232];
/******************4*******************/
assign DOUT[75] = DIN[154] ^ DIN[117];
assign DOUT[77] = DIN[155] ^ DIN[118];
assign DOUT[79] = DIN[156] ^ DIN[119];
…
assign DOUT[231] = DIN[232] ^ DIN[195];
/******************5*******************/
assign DOUT[148] = DIN[74];
assign DOUT[150] = DIN[75];
assign DOUT[152] = DIN[76];
…
```

```
    assign DOUT[232] = DIN[116];
    endmodule
```

平方模块是一个纯组合逻辑电路，用 Verilog HDL 语言进行描述，将椭圆曲线的二进制表示固化在程序当中，以减少输入端口。

6.2.4　有限域模逆的 FPGA 实现

在 ECC 中有一个重要的运算是两个多项式的除法，而除法又是通过乘以乘法逆元来实现的。比较有名的计算乘法逆元的方法有两种：基于费尔马定理的模逆算法和扩展欧几里德算法，在这里，用基于扩展欧几里德算法的模逆计算算法。

令 $F_{2^m}^*$ 表示 F_{2^m} 中所有非零元素的集合，则因为定义多项式 $f(x)$ 不可约，所以任意一个非零元素 $a \in F_{2^m}^*$ 对应的多项式 $A(x)$ 都有 $\gcd(f(x), A(x)) = 1$，其中 gcd() 表示最大公因式，于是存在多项式 $B(x)$ 和 $C(x)$，使得 $A(x) \cdot B(x) = f(x) \cdot C(x) + 1$，即 $A(x) \cdot B(x) = 1 \bmod (f(x))$ 所对应的元素 $b = a^{-1}$，称为元素 a 的逆，$B(x)$ 可以通过扩展的欧几里德算法得到。

模逆运算需要调用乘法模块，由于椭圆曲线密码算法中的模逆与乘法运算是分步进行的，这样可以将乘法器集成在模逆运算单元中，组成一个乘/逆运算单元。模逆运算的程序实现如程序 6-4 所示。

程序 6-4：模逆模块

```
    module MODINV_TOP(
                      CLK,
                      RST_N,
                      DIN,
                      DOUT,
                      IN_VALID,
                      OUT_VALID
                      );
                      …
    always @(posedge CLK)
        if(!RST_N)
            begin
            DIN_temp1 <= 0;
            OUT_VALID <= 1'b0;
            cnt <= 0;
            end
        else if(IN_VALID)
            begin
            DIN_temp1 <= DIN;
            cnt <= cnt + 1;
            end
        else if(cnt == 3)
            begin
            …
```

```
end
always @(posedge CLK)
    if(OUT_VALID == 1'b1)
        begin
        DOUT <= DIN;
        OUT_VALID <= 1'b1;
        end
    else
        begin
        DOUT <= DIN;
        OUT_VALID <= 1'b0;
        end
Endmodule
```

程序 6-5：状态机程序

```
//result
module MODINV_FSM(
                CLK,
                RST_N,
                TLoad,
                TClear,
                XLoad,
                XClear,
                ASel,
                BSel,
                OUT_VALID_tmp,
                IN_VALID_tmp,
                OUT_STATE
                );
        …
always @(posedge CLK)
    if(!RST_N)
        begin
        cState <= {5{1'b0}};
        state6_reg <= {2{1'b0}};
        state11_reg <= {3{1'b0}};
        state14_reg <= {4{1'b0}};
        state19_reg <= {5{1'b0}};
        state22_reg <= {6{1'b0}};
        state25_reg <= {7{1'b0}};
        OUT_STATE <= {5{1'b0}};
        end
```

```
        else
            begin
            cState <= nState;
            OUT_STATE <= cState;
            end
    always @(cState)
        case(cState)
            5'b00000 : begin  //initial state
                        TLoad = 0; TClear = 1; XLoad = 0; XClear = 1;
                        ASel = 0; BSel = 0;
                        nState = 5'b00001;
                       end
            5'b00001 : begin //first state   1:T=A^2
                        TLoad = 1; TClear = 0; XLoad = 1; XClear = 0;
                        ASel = 0; BSel = 1;
                        IN_VALID_tmp<=1'b1; //multi input valid !
                        nState = 5'b00010;
                       end
            5'b00010 : begin // 2:X=AT
                        TLoad = 1; TClear = 0; XLoad = 1; XClear = 0;
                        ASel = 0; BSel = 1;
                        if(OUT_VALID_tmp == 1'b1)  //multi operation finish !
                                begin
                                IN_VALID_tmp <= 1'b1;
                                nState = 5'b00011;
                                end
                        else
                                begin
                                nState = 5'b00010;
                                IN_VALID_tmp <= 1'b0;
                                End
            …
            default : begin
                        TLoad = 0; TClear = 0; XLoad = 0; XClear = 0;
                        ASel = 0; BSel = 0;
                        IN_VALID_tmp <= 1'b0;
                        nState = 5'b00000;
                      end
            endcase
         end
```

在程序 6-5 中，用 case 语句来运算状态，如果 cState==5'b00000，对状态进行初始化；如果 cState==5'b00001，进行模方运算；如果 cState==5'b00010，进行模方运算。

6.2.5 点加和倍加的 FPGA 实现

对于特征值为 2 的有限域 F_{2^m} 上的加法和倍乘的运算规则如下。

椭圆曲线 $E(F_{2^m})$ 上的加法规则为

$$x_3=\lambda^2+\lambda+x_1+x_2+a,\ y_3=\lambda(x_1+x_3)+x_3+y_1$$

其中 $\lambda=\dfrac{y_1+y_2}{x_1+x_2}$，$P$=($x_1$，$y_1$)，$Q$=($x_2$，$y_2$)，$W$=($x_3$，$y_3$)=$P$+$Q$，$P$，$Q$，$W$ 为椭圆曲线上的 3 个点。

$E(F_{2^m})$ 上的 2 倍加规则为：$x_3=\lambda^2+\lambda+a,\ y_3=x_1^2+(1+\lambda)x_3$

其中 $\lambda=x_1+\dfrac{y_1}{x_1}$，$P$=($x_1$，$y_1$)，$W$=($x_3$，$y_3$)=2$P$，$P$，$W$ 为椭圆曲线上的 2 个点。

点加和倍加运算的定义虽然不同，但是它们的主要计算开销（硬件资源和运算时间）是一致的，而且在群预算的过程中，点加和倍加运算不能同时进行，因此没有必要设计单独的点加模块与倍加模块，可以将它们作为一个整体来实现。本节将给出点加和 2 倍加算法的具体实现。实现的程序如程序 6-6 所示。

程序 6-6：点加和倍加模块

```
module PAdd_TOP(
                CLK,
                RST_N,
                DIN_P0_x,
                DIN_P0_y,
                DIN_P1_x,
                DIN_P1_y,
                DOUT_P2_x,
                DOUT_P2_y,
                IN_VALID,
                OUT_VALID
)
…
PAdd_FSM pa_fsm(
                .CLK(CLK),
                .RST_N(RST_N),
                .P0_x_Load(P0_x_Load),
                .P0_x_Clear(P0_x_Clear),
                .P0_y_Load(P0_y_Load),
                .P0_y_Clear(P0_y_Clear),
                .P1_x_Load(P1_x_Load),
                .P1_x_Clear(P1_x_Clear),
                .P1_y_Load(P1_y_Load),
                .P1_y_Clear(P1_y_Clear),
```

```
                .reg_add1_out_Load(add1_out_Load),
                .reg_add1_out_Clear(add1_out_Clear),
                .INV_IN_VALID(INV_IN_VALID),
                .INV_OUT_VALID(INV_OUT_VALID),
                .reg_inv_out_Load(inv_out_Load),
                .reg_inv_out_Clear(inv_out_Clear),
                .reg_add2_out_Load(add2_out_Load),
                .reg_add2_out_Clear(add2_out_Clear),
                .reg_mult1_out_Load(mult1_out_Load),
                .reg_mult1_out_Clear(mult1_out_Clear),
                .reg_mult2_out_Load(mult2_out_Load),
                .reg_mult2_out_Clear(mult2_out_Clear),
                .reg_add5_out_Load(add5_out_Load),
                .reg_add5_out_Clear(add5_out_Clear),
                .MULT1_IN_VALID(mult1_IN_VALID),
                .MULT1_OUT_VALID(mult1_OUT_VALID),
                .MULT2_IN_VALID(mult2_IN_VALID),
                .MULT2_OUT_VALID(mult2_OUT_VALID),
                .AP_OUT_STATE(ap_OUT_STATE)
                );
    always @(posedge CLK)
        if(!RST_N)
            begin
            DIN_P0_x_temp <= 0;
            DIN_P0_y_temp <= 0;
            DIN_P1_x_temp <= 0;
            DIN_P1_y_temp <= 0;
            cnt_top <= 0;
            end

        else if(IN_VALID)
            begin
            DIN_P0_x_temp <= DIN_P0_x;
            DIN_P0_y_temp <= DIN_P0_y;
            DIN_P1_x_temp <= DIN_P1_x;
            DIN_P1_y_temp <= DIN_P1_y;
            end
        else
            begin
            DIN_P0_x_temp <= 0;
            DIN_P0_y_temp <= 0;
            DIN_P1_x_temp <= 0;
            DIN_P1_y_temp <= 0;
            cnt_top <= cnt_top + 1;
```

```
        end
    always @(posedge CLK)
        if((ap_OUT_STATE == 3'b100) && (mult2_OUT_VALID == 1'b1))
            begin
            DOUT_P2_x <= DOUT_add4;
            DOUT_P2_y <= DOUT_add7;
            OUT_VALID <= 1'b1;
            end
        else if(cnt_top == 13)
            begin
            ……
            end
        else
            begin
            DOUT_P2_x <= 0;
            DOUT_P2_y <= 0;
            OUT_VALID <= 1'b0;
            end
    endmodule
```

程序 6-7：状态控制机程序

```
    module PAdd_FSM(
                    CLK,
                    RST_N,
                    P0_x_Load,
                    P0_x_Clear,
                    P0_y_Load,
                    P0_y_Clear,
                    P1_x_Load,
                    P1_x_Clear,
                    P1_y_Load,
                    P1_y_Clear,
                    reg_add1_out_Load,
                    reg_add1_out_Clear,
                    INV_IN_VALID,
                    INV_OUT_VALID,
                    reg_inv_out_Load,
                    reg_inv_out_Clear,
                    reg_add2_out_Load,
                    reg_add2_out_Clear,
                    reg_mult1_out_Load,
                    reg_mult1_out_Clear,
                    reg_mult2_out_Load,
                    reg_mult2_out_Clear,
```

```
                reg_add5_out_Load,
                reg_add5_out_Clear,
                MULT1_IN_VALID,
                MULT1_OUT_VALID,
                MULT2_IN_VALID,
                MULT2_OUT_VALID,
                AP_OUT_STATE
                );
…
always @(posedge CLK)
    if(!RST_N)
        begin
        cState = 0;
        AP_OUT_STATE = 0;
        end
    else
        begin
        cState = nState;
        AP_OUT_STATE = cState;
        end
always @(cState)
    case(cState)
    3'b000 : begin //initial state
        INV_IN_VALID = 1;
        MULT1_IN_VALID = 0; MULT2_IN_VALID = 0;
        P0_x_Load = 0; P0_x_Clear = 1; P0_y_Load = 0; P0_y_Clear = 1;
        P1_x_Load = 0; P1_x_Clear = 1; P1_y_Load = 0; P1_y_Clear = 1;
        reg_add1_out_Load = 0; reg_add1_out_Clear = 1;
        reg_inv_out_Load = 0; reg_inv_out_Clear = 1;
        reg_add2_out_Load = 0; reg_add2_out_Clear = 1;
        reg_mult1_out_Load = 0; reg_mult1_out_Clear = 1;
        reg_mult2_out_Load = 0; reg_mult2_out_Clear = 1;
        reg_add5_out_Load = 0; reg_add5_out_Clear = 1;
        nState = 3'b001;
        end
    3'b001 : begin //1. inv
        //INV_IN_VALID = 1;
        MULT1_IN_VALID = 0; MULT2_IN_VALID = 0;
        P0_x_Load = 1; P0_x_Clear = 0; P0_y_Load = 0; P0_y_Clear = 1;
        P1_x_Load = 1; P1_x_Clear = 0; P1_y_Load = 0; P1_y_Clear = 1;
        reg_add1_out_Load = 1; reg_add1_out_Clear = 0;
        reg_inv_out_Load = 0; reg_inv_out_Clear = 1;
        reg_add2_out_Load = 0; reg_add2_out_Clear = 1;
```

```
            reg_mult1_out_Load = 0; reg_mult1_out_Clear = 1;
            reg_mult2_out_Load = 0; reg_mult2_out_Clear = 1;
            reg_add5_out_Load = 0; reg_add5_out_Clear = 1;
                        if(INV_OUT_VALID)
                            begin
                            INV_IN_VALID = 0;
                            P0_y_Load = 1; P0_y_Clear = 0;
                            P1_y_Load = 1; P1_y_Clear = 0;
                            reg_add2_out_Load = 1; reg_add2_out_Clear = 0;
                            reg_inv_out_Load = 1; reg_inv_out_Clear = 0;
                            MULT1_IN_VALID = 1;

                            nState = 3'b010;
                            end
                        else
                            begin
                            INV_IN_VALID = 0;
                            nState = 3'b001;
                            end
                    end
        3'b010 : …
        3'b011 : …
    endmodule
```

本模块中将点加和倍加运算单元做成一个模块，首先对状态进行初始化，采用点加和倍加运算状态机对有限域运算单元进行控制，分别完成点加和倍加运算。

6.2.6 点乘的 FPGA 实现

在椭圆曲线密码体制中，有限域乘法是最基本的运算，设计快速的乘法器对于提高椭圆曲线密码体制的加解密速度非常重要。关于 GF_{2^m} 域乘法器的设计已有不少思路，根据运算数的形式分别有正规基、多项式基和双基，由于多项式基是实现不同基之间进行转换的基础，因此主要研究多项式基上乘法运算的设计与实现。

已经提出多种有限域乘法器的优化算法，但主要是根据处理器或 ASIC 芯片的特点进行分析而提出的，而针对 FPGA 芯片的特点进行分析和优化的算法较少。与 ASIC 芯片的 2 输入门的细粒度结构不同，FPGA 芯片主要是一种基于 4 输入查找表（LUT）、具有高输入输出能力的逻辑单元结构。使用查找表的好处是任意 4 输入真值表都可以实现，资源利用率较高，所以 FPGA 芯片具有编程方便、集成度高、速度快等特点。FPGA 芯片与 ASIC 芯片最大的区别在于，FPGA 芯片可以由用户编程，而 ASIC 芯片必须在工厂制作，花费的时间较多。

根据 NIST 推荐的 5 个二进制有限域 $F_{2^{163}}, F_{2^{233}}, F_{2^{283}}, F_{2^{409}}, F_{2^{571}}$，通常认为 m 大于 160 比特以上时，椭圆曲线密码体制才具有足够的安全性，以下给出了一种 m 为 233 的情况下多项式基乘法器的实现结构，并根据不同的步长给出了相应数据。

实现多项式相乘最基本的方法就是移位相加算法，该算法需要大量的移位操作，不适合基于处理器的实现方案。研究者们相继提出的梳状算法、梳状窗口算法以及相应的约简算法等，思路是把乘法和求余分开计算，即首先计算两个多项式的乘积，再用不可约多项式约简。但软件实现速度较慢，而费时的移位操作在硬件中实现起来比较容易，并且不占用任何逻辑资源，因此该算法非常适合用于 FPGA 实现。按照硬件实现的结构来分，多项式基乘法运算可以分为基于比特的全串行结构、全并行结构以及串并混合结构。全并行结构要求在一个时钟周期内中完成全部的计算过程，硬件实现的开销大，只具有理论研究的意义；全串行结构是在每个时钟周期只完成一次迭代计算，需要很多个时钟周期。这里采用串并混合结构模式，既解决了硬件开销问题，又解决了速度问题。

具体实现代码如程序 6-8 和 6-9 所示。

程序 6-8：点乘模块算法

```
module PointMult_FSM(
                    CLK,
                    RST_N,
                    KEY_EQ_1,
                    Sel1,
                    Sel2,
                    reg_s1_out_Load,
                    reg_s1_out_Clear,
                    reg_GP_Load,
                    reg_GP_Clear,
                    reg_DQ_out_Load,
                    reg_DQ_out_Clear,
                    reg_PA_out_Load,
                    reg_PA_out_Clear,
                    DQ_IN_VALID,
                    DQ_OUT_VALID,
                    PointAdd_IN_VALID,
                    PointAdd_OUT_VALID,
                    PM_COUNT,
                    PM_OUT_STATE
                    );
…
always @(posedge CLK)
    if(!RST_N)
        begin
        PM_COUNT <= 8'b00000000;
        cState = 0;
        PM_OUT_STATE = 0;
        end
    else
        begin
```

```
            cState = nState;
            PM_OUT_STATE = cState;
            PM_COUNT <= PM_COUNT_temp;
            end
    always @(cState)
        case(cState)
            2'b00 : begin //initial state
                    Sel1 = 1;
                    reg_s1_out_Load = 1;
                    reg_s1_out_Clear = 0;
                    DQ_IN_VALID = 1;
                    PM_COUNT_temp <= PM_COUNT_temp + 8'b00000001;
                    nState = 2'b01;
                    end

            2'b01 : …

            2'b10 : …
            default : begin
                        nState = 2'b00;
                      end
            endcase
    endmodule
```

程序 6-9：点乘算法状态机模块

```
    module PointMult_FSM(
                        CLK,
                        RST_N,
                        KEY_EQ_1,
                        Sel1,
                        Sel2,
                        reg_s1_out_Load,
                        reg_s1_out_Clear,
                        reg_GP_Load,
                        reg_GP_Clear,
                        reg_DQ_out_Load,
                        reg_DQ_out_Clear,
                        reg_PA_out_Load,
                        reg_PA_out_Clear,
                        DQ_IN_VALID,
                        DQ_OUT_VALID,
                        PointAdd_IN_VALID,
                        PointAdd_OUT_VALID,
                        PM_COUNT,
```

```
                    PM_OUT_STATE
    …
    always @(posedge CLK)
        if(!RST_N)
            begin
            PM_COUNT <= 8'b00000000;
            cState = 0;
            PM_OUT_STATE = 0;
            end
        else
            begin
            cState = nState;
            PM_OUT_STATE = cState;
            PM_COUNT <= PM_COUNT_temp;
            end
    always @(cState)
        case(cState)
            2'b00 : begin //initial state
                    Sel1 = 1;
                    reg_s1_out_Load = 1;
                    reg_s1_out_Clear = 0;
                    DQ_IN_VALID = 1;
                    PM_COUNT_temp <= PM_COUNT_temp + 8'b00000001;
                    nState = 2'b01;
                    end
            2'b01 : …
            2'b10 : …
            default : begin
                        nState = 2'b00;
                      end
            endcase
    endmodule
```

由于全串行结构的乘法器实现速度慢，较好的办法就是根据芯片的资源情况采用串并混合结构的实现方案。FPGA 芯片每个时钟完成 K 次迭代计算，即在同一个时钟内完成 K 次累加、左移和约简的过程。此时需要的时钟数为$[m/k]+1$ 个，乘法运算的速度将获得成倍的提高。

6.3 ECC 算法工程实现

1．创建 ISE 工程

打开 ISE 开发环境，选择菜单 File→New Project，建立一个新工程，然后设置顶层文件模块名、存储目录和设计方式，如图 6.2 所示。接着选择器件，这里选择 Xinlix 的 5vlx20tff323

芯片来实现。再选择仿真工具和编程语言，这里选择第三方仿真软件 ModelSim 和 Verilog 语言，如图 6.3 所示。最后完成新工程的建立。

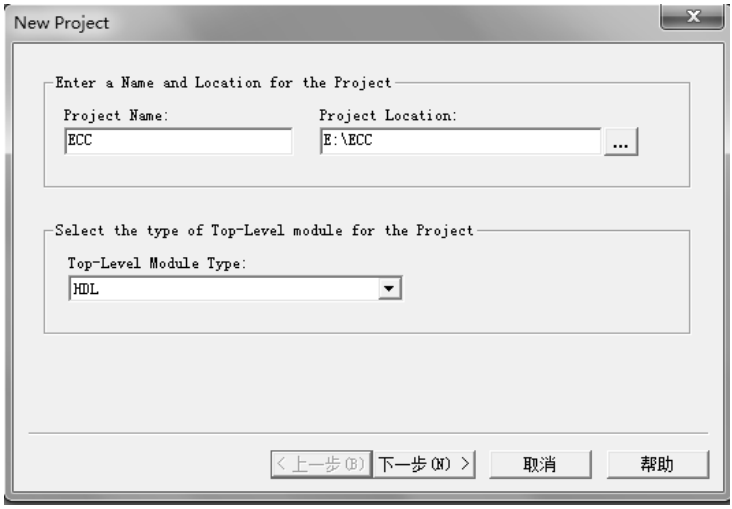

图 6.2　ECC 算法工程文件的建立

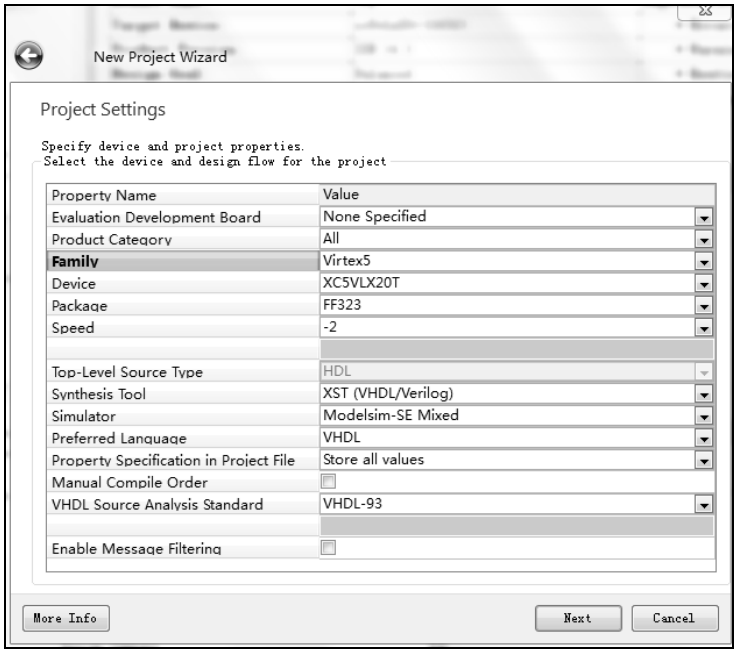

图 6.3　器件的选择和工具的使用

2．编写设计代码

将 6.2 节中编写的模块代码导入新建的工程中，具体实现如图 6.4 所示。

3．工程验证

（1）根据以上编写的程序，编写该程序的测试文件，如图 6.5 所示。

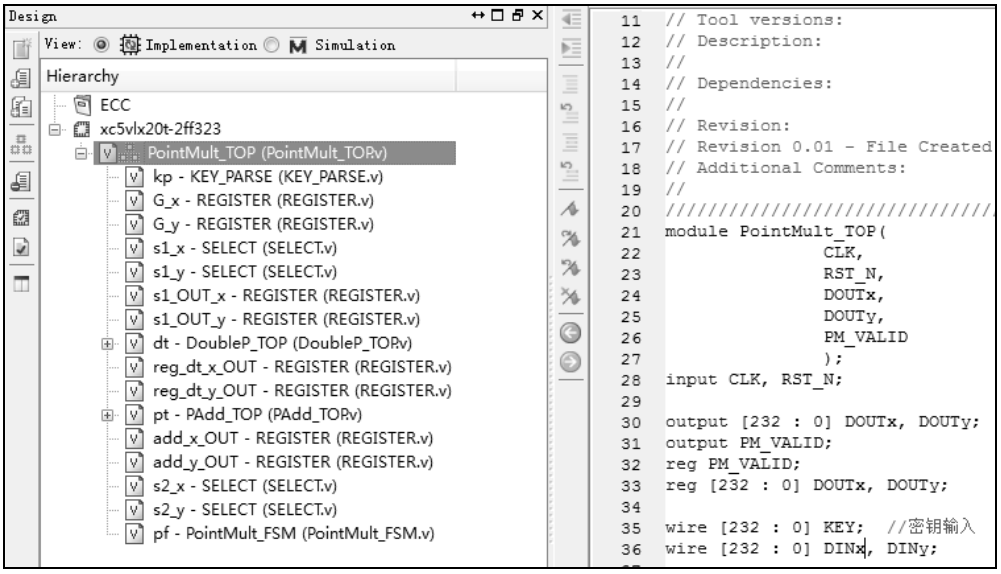

图 6.4　ECC 算法的 FPGA 代码实现

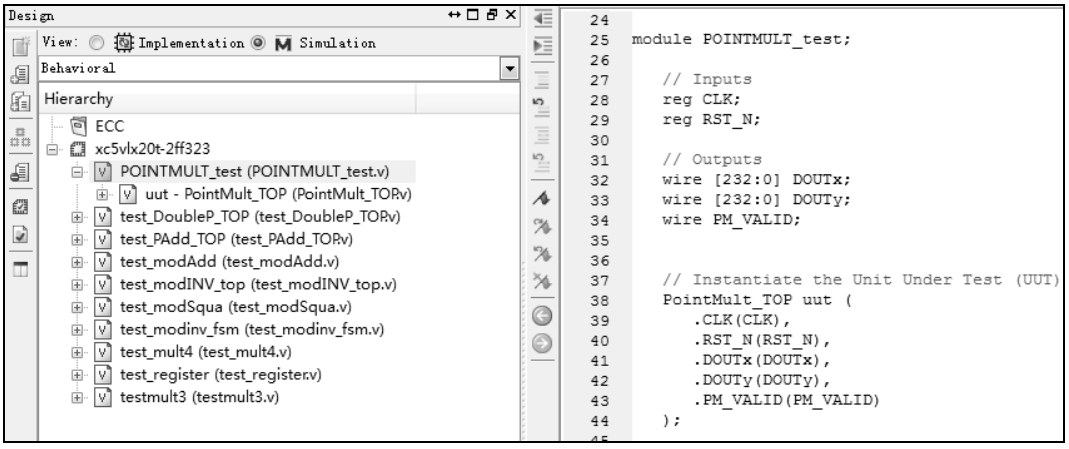

图 6.5　ECC 算法测试文件

（2）根据上一步的测试文件，利用 ModelSim 仿真软件，对该算法进行仿真测试，如图 6.6 所示，仿真结果如图 6.7 所示。

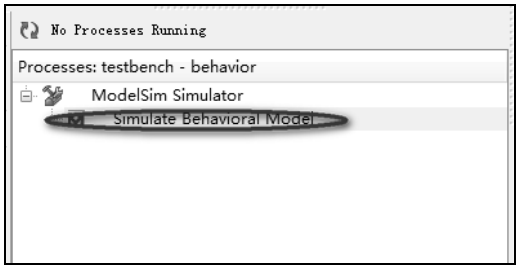

图 6.6　ModelSim 调用示意图

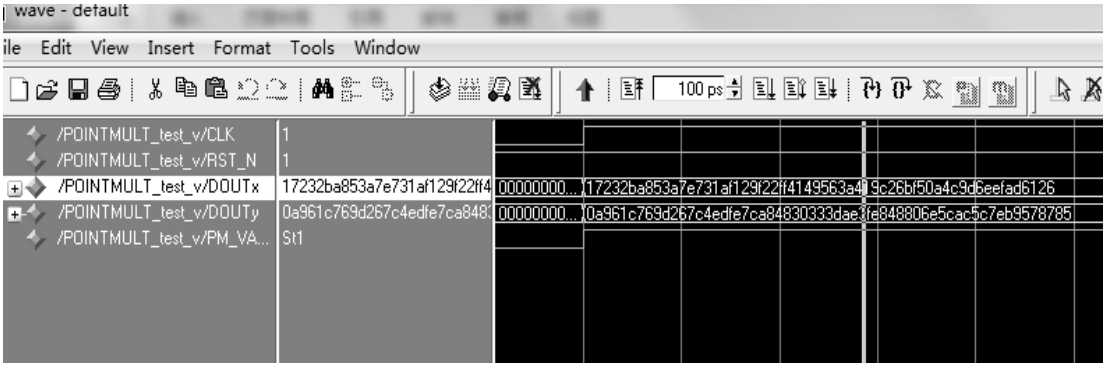

图 6.7 ECC 算法中点乘模块仿真结果

6.4 效果测试

为了验证本 ECC 算法中点乘算法设计的正确性，下面编写了测试程序，用 ModelSim SE10.1 仿真工具进行仿真测试，该测试所使用的电脑配置为：四核 i5-3470 CPU，4G 内存，32 位操作系统。

所选点的坐标及椭圆参数如下。

点 x：17232BA853A7E731AF129F22FF4149563A419C26BF50A4C9D6EEFAD6126；

点 y：1DB537DECE819B7F70F555A67C427A8CD9BF18AEB9B56E0C11056FAE6A3；

k： 8000000000000000000000000000069D5BB915BCD46EFB1AD5F173ABDE。

得到点乘的点坐标如下。

点 x：17232BA853A7E731AF129F22FF4149563A419c26BF50A4C9D6EEFAD6126；

点 y：0A961C769D267C4EDFE7CA84830333DAE3FE848806E5CAC5C7EB9578785；

计算点乘所用时间为 3330 us。

下面对该工程的时序、运行效率、资源占用情况进行分析。在 Xilinx 公司的 5vlx20tff323 元器件上，运用 ISE 的 XST 分析工具进行综合后的结果如图 6.8 所示。

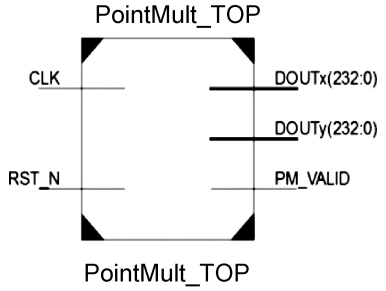

图 6.8 ECC 算法点乘运算综合后的外部器件

资源占用情况分析报告如表 6.1 所示。

表 6.1　ECC 算法点乘运算资源占用情况

Selected Device : 5vlx20tff323-2				
Number of Slice Registers	45	out of	12480	0%
Number of Slice LUTs	51	out of	12480	0%
Number used as Logic	51	out of	12480	0%
Number of LUT Flip Flop pairs used	62			
Number with an unused Flip Flop	17	out of	62	27%
Number with an unused LUT	11	out of	62	17%
Number of fully used LUT-FF pairs	34	out of	62	54%
Number of unique control sets	12			

时序分析报告如下。

Maximum Frequency: 373.769 MHz

由综合报告功能仿真可知，ECC 算法的点乘运算速率可达到 300 次/秒。

6.5 本章小结

本章结合椭圆曲线算法的数学基础对椭圆曲线密码体制进行比较深入的分析，对椭圆曲线密码算法的 FPGA 实现进行了具体的研究设计。椭圆曲线密码算法的实现包括底层的有限域运算和上层的群运算，有限域乘法控制模块、有限域加法控制模块、有限域平方控制模块、有限域模逆控制模块；点加和倍加控制模块、点乘状态机控制模块等。

椭圆曲线密码体制是目前应用广泛的公钥密码体制，椭圆曲线密码系统由于其自身的优点，在今后一段时间内还会有更大的发展空间，其发展趋势有两个方面：一是对椭圆曲线密码算法的改进，椭圆曲线上域运算相当丰富，将来应该出现更简洁更适于硬件实现的算法；二是硬件器件的改进，对于如 FPGA 之类的器件，提高其综合和实现效率，对椭圆曲线密码系统运算速度的提高和硬件面积的减少会有很大的作用。本章讲述了一种椭圆曲线点乘算法的 FPGA 实现方法，并介绍了执行速度、资源占用以及参数配置等几个方面，希望读者以后能做进一步的改进和优化。

第 7 章　SM2 算法 FPGA 实现

2012 年 12 月，国家密码管理局公布了 SM2 椭圆曲线公钥密码算法。SM2 是我国自主知识产权的商用密码算法，是 ECC 算法的一种，其加密强度为 256 位。SM2 椭圆曲线公钥密码算法基于椭圆曲线离散对数问题，计算复杂度是指数级的，求解难度较大。在相同安全程度要求下，椭圆曲线密码算法较其他公钥密码算法所需密钥长度要小得多。详细内容可以参考国家密码管理局公布的《SM2 椭圆曲线公钥密码算法》白皮书。本章将介绍 SM2 密码体制的算法流程、密钥生成、加密过程和解密过程，以及 FPGA 程序实现。

7.1　算法原理

7.1.1　密钥生成

选择一个椭圆曲线 $E_p(a,b): y^2 = x^3 + ax + b (\mathrm{mod}\ p)$，构造一个椭圆群 $E_p(a,b)$。

在椭圆群 $E_p(a,b)$ 中挑选生成元点 $G=(x_0,y_0)$，G 为使得满足 $nG=O$ 的最小的 n，是一个非常大的素数（N 表示椭圆群 $E_p(a,b)$ 的元素个数，记为：$N=\#E(F_p)$，n 是 N 的素因子）。

用随机数发生器产生随机数 $d(1<d<\#E(F_p))$ 作为私钥，相应的公钥为 $P=[d]\cdot G=(x_p,y_p)$。可知，由公开密钥 P 求解私钥 d 是困难的，因为要求解椭圆曲线离散对数问题（Elliptic Curve Discrete Logarithm Problem，ECDLP）。SM2 算法的公钥为 (P,G)，私钥为 d。

7.1.2　加密过程

假设用户 A 要发送给用户 B 的消息为比特串 M，len 为 M 的比特长，用户 A 要进行如下的加密运算。这里，用户 A 和用户 B 的公私钥对分别为 $(P_\mathrm{A},d_\mathrm{A})$ 和 $(P_\mathrm{B},d_\mathrm{B})$。

第一步：用随机数发生器产生随机数 $k\in[1,n-1]$，其中 n 是椭圆曲线基点 G 的阶次。

第二步：计算椭圆曲线点 $C_1=[k]G=(x_1,\ y_1)$，其中 G 和 C_1 都在椭圆曲线上，并将 C_1 的数据类型转换成比特串。

第三步：计算椭圆曲线上的点 $S=[h]P_\mathrm{B}$，$h\in[1,n-1]$，若 S 是无穷远点，则报错并退出。

第四步：计算曲线点 $[k]P_\mathrm{B}=(x_2,\ y_2)$，将坐标 x_2、y_2 的数据类型转换成比特串。

第五步：计算 $t=\mathrm{KDF}(x_2\|y_2,\mathrm{len})$，KDF() 是密钥派生函数，输出长度是 len 的比特串，若 t 为全零比特串，则返回第一步。

第六步：计算 $C_2=M\oplus t$。

第七步：计算 $C_3=\mathrm{Hash}(x_2\|M\|y_2)$，Hash() 是密码杂凑函数。

第八步：输出密文 $C=C_1\|C_2\|C_3$，并发送给用户 B。

SM2 算法加密流程如图 7.1 所示。

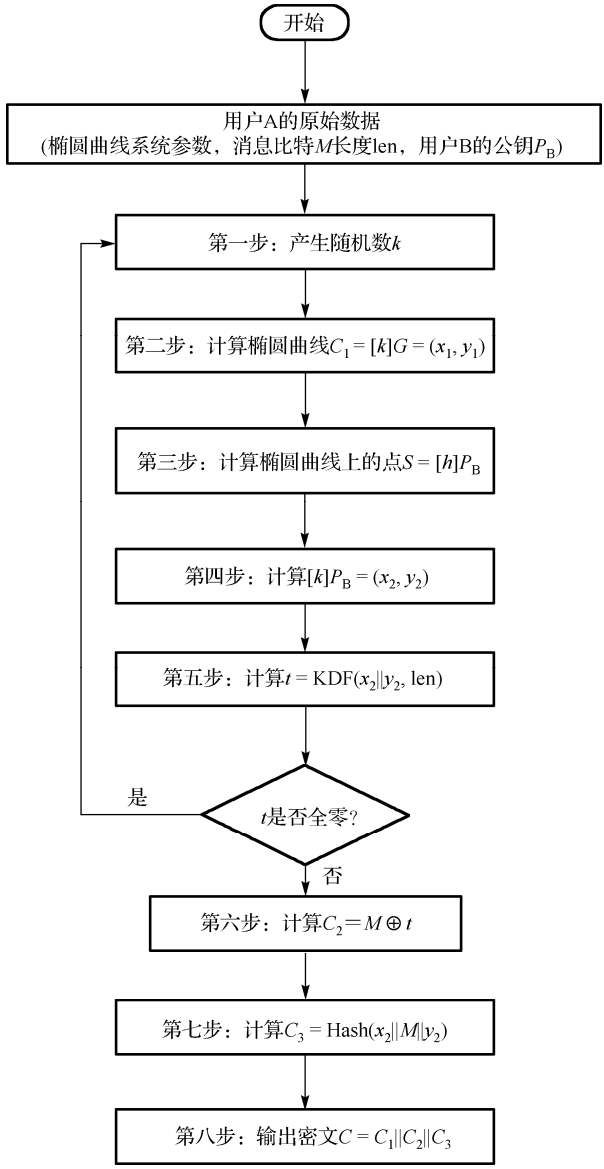

图 7.1　SM2 加密算法流程

7.1.3　解密过程

假设 len 为密文中 C_2 的比特长度，用户 B 要对密文 $C = C_1 \| C_2 \| C_3$ 进行解密，需要完成以下运算。

第一步：从 C 中取出比特串 C_1，将 C_1 的数据类型转换为椭圆曲线上的点，验证 C_1 是否满足椭圆曲线方程，若不满足则报错并退出。

第二步：计算椭圆曲线点 $S = [h]G$，若 S 是无穷远点，则报错并退出。

第三步：计算 $[d_B]C_1 = (x_2, y_2)$，将坐标 x_2、y_2 的数据类型转换为比特串。

第四步：计算 $t = \mathrm{KDF}(x_2 \| y_2, \mathrm{len})$，若 t 为全零比特串，则报错并退出。

第五步：从 C 中取出比特串 C_2，计算 $M' = C_2 \oplus t$。

第六步：计算 $u = \mathrm{Hash}(x_2 \| M' \| y_2)$，从 C 中取出比特串 C_3，若 $u \neq C_3$，则报错并退出。

第七步：用户 B 输出明文 M'。

SM2 算法解密过程流程图如图 7.2 所示。

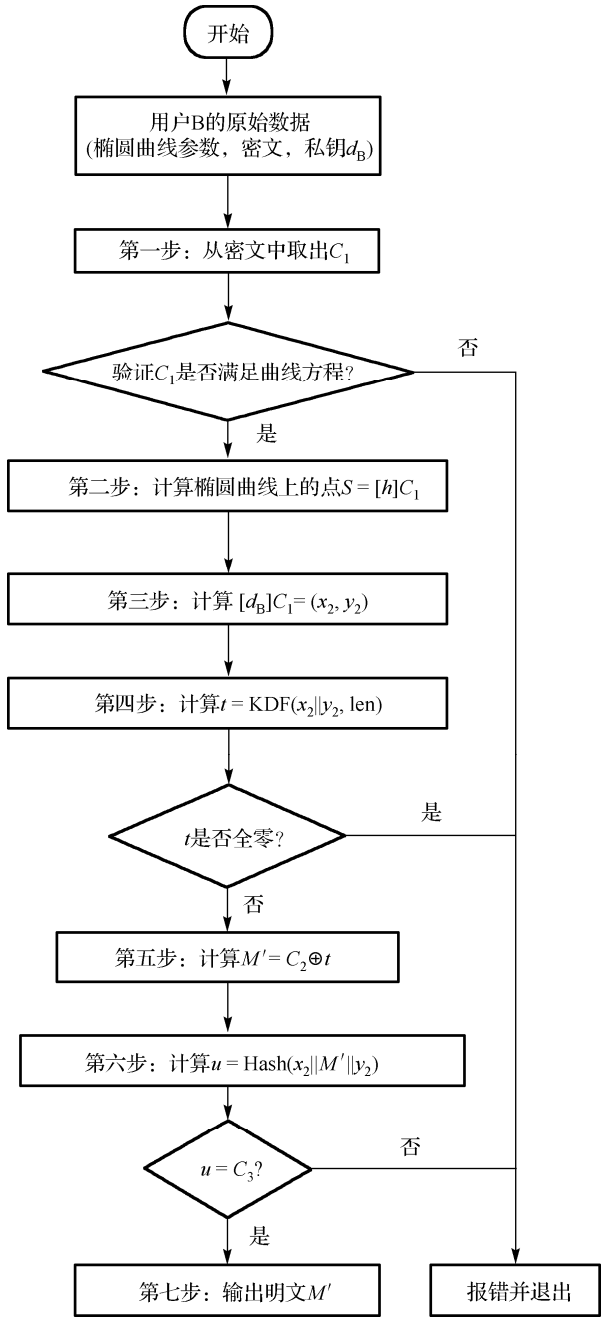

图 7.2　SM2 算法解密流程

7.2 SM2 算法相关模块 FPGA 设计

通过对算法结构的分析发现，用 FPGA 实现该算法主要需要以下几个模块：点乘模块、Hash 算法模块、点加模块、倍点模块、坐标转换模块、模逆运算模块和模乘模块。其中，倍点乘运算模块是整个算法的核心，它影响着整个算法的实现速率。系统设计如图 7.3 所示。

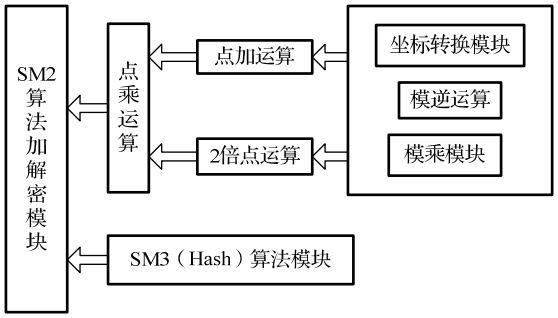

图 7.3　SM2 算法模块图

7.2.1 坐标转换模块设计

椭圆曲线上的点有几种不同的坐标表示方式，不同的坐标表示在计算点的加法和两倍点时运算效率不一样。通常，椭圆曲线上点的表示方式主要有三种：齐次坐标、仿射坐标和 Jacobi 坐标，由于齐次坐标在实际中应用不多，因此，只考虑在仿射坐标和 Jacobi 坐标中点的运算。

仿射坐标：用 (x,y) 表示椭圆曲线上的点，无穷点用 o 表示。它与 Jacobi 坐标的关系是

$$X=x,\ \ Y=y,\ \ Z=1$$

Jacobi 坐标：用 (X,Y,Z) 表示椭圆曲线上的点，它与仿射坐标的关系是

$$x=X/Z^2,\ \ y=Y/Z^3$$

其中 $(0,1,0)$ 表示无穷远点。

由上面的分析可知，从仿射坐标到 Jacobi 坐标，只需要令 $X=x,\ \ Y=y,\ \ Z=1$ 即可，实现较简单。而从 Jacobi 坐标到仿射坐标，需要三次乘法运算，一次平方运算和一次求逆运算。

7.2.2 点加运算和 2 倍点运算设计

素数域 GF(P)仿射坐标系下的椭圆曲线 $y^2=x^3+ax+b$ 在 Jacobi 加重射影坐标系下对应的曲线方程为 $Y^2=X^3+aXZ^4+bZ^6$。仿射坐标系下的点 (x,y)（非无穷远点）在 Jacobi 加重射影坐标系下对应的点为 $(x,y,1)$，无穷远点对应的点是 $(1,1,0)$；Jacobi 加重射影坐标系下的点 (X,Y,Z)（非无穷远点，故 $Z\neq 0$）在仿射坐标系下对应的点是 $(X/Z^2,Y/Z^3)$，$Z=0$ 时对应仿射坐标系下的无穷远点。

Jacobi 加重射影坐标系下点的加法计算公式如下。

设 $P=(X_1,Y_1,Z_1)$，$Q=(X_2,Y_2,Z_2)$，$R=(X_3,Y_3,Z_3)=P+Q$，如果 $P\neq Q$，则：

$$\lambda_1 = X_1 Z_2^2,\ \lambda_2 = X_2 Z_1^2,\ \lambda_3 = \lambda_1 - \lambda_2,$$
$$\lambda_4 = Y_1 Z_2^3,\ \lambda_5 = Y_2 Z_1^3,\ \lambda_6 = \lambda_4 - \lambda_5,$$
$$\lambda_7 = \lambda_1 + \lambda_2,\ \lambda_8 = \lambda_4 + \lambda_5,$$
$$Z_3 = Z_1 Z_2 \lambda_3,\ X_3 = \lambda_6^2 - \lambda_7 \lambda_3^2,$$
$$\lambda_9 = \lambda_7 \lambda_3^2 - 2X_3,\ Y_3 = (\lambda_9 \lambda_6 - \lambda_8 \lambda_3^3)/2$$

若 $P = Q$，则可以得到两倍加的计算公式：

$$\lambda_1 = 3X_1^2 + aZ_1^4,\ Z_3 = 2Y_1 Z_1,$$
$$\lambda_2 = 4X_1 Y_1^2,\ X_3 = \lambda_1^2 - 2\lambda_2,$$
$$\lambda_3 = 8Y_1^4,\ Y_3 = \lambda_1(\lambda_2 - X_3) - \lambda_3$$

其中 a 是椭圆曲线的参数。

在具体实现点加运算时，除了判断相加的两个点是否为无穷远点外，还要判断这两个点是否是同一个点，以确定是用点加公式（$P \neq Q$ 的情形）求和还是用倍点公式（$P = Q$ 的情形）求和。由于在 Jacobi 加重射影坐标系下 (X, Y, Z) 和 (u^2X, u^3Y, uZ) 表示同一个点，这里 $u \in \mathrm{GF}(P)$ 且 $u \neq 0$，因此在具体判断两个点是否为同一点时不能简单地采用坐标比较的方法。

7.2.3 点乘运算设计

椭圆曲线上同一个点的多次加称为该点的点乘运算。设 k 是一个正整数，P 是椭圆曲线上的点，称点 P 的 k 次加为点 P 的 k 倍点运算，记为 $Q = [k]P = P + \cdots + P$，等式最右边的部分包括 k 个 P。设素数域 $\mathrm{GF}(P)$ 上的椭圆曲线 $y^2 = x^3 + ax + b$ 上的所有点加上无穷远点构成的集合为 $E(\mathrm{GF}(P))$，由于 $E(\mathrm{GF}(P))$ 在定义的点加运算下构成一个阿贝尔群，椭圆曲线点乘运算可以归结为基本的点加运算。在仿射坐标系下每次基本的点加运算都会用到素数域 $\mathrm{GF}(P)$ 上的求逆运算，由于素数域上的求逆运算相对于其它的运算非常耗费时间，这样就大大影响了点乘运算的运行速度。因此在进行多倍点运算的算法实现时，采用在 Jacobi 加重射影坐标系下进行，最后将计算的结果转化到仿射坐标系下。Jacobi 加重射影坐标系下的点加运算不包含求逆运算，这样就可通过增加几个模乘运算，加上最后坐标转化时的一个求逆运算，便可完成多倍点的计算，从而提高了多倍点运算的运行效率。

SM2 算法中点乘运算归结为基本的点加运算，实现算法常见的有从右到左的二进制法、从左到右的二进制法、加减法（NAF 算法）、滑动窗口法等。

采用了从左到右的二进制法，可以有效减少多倍点运算中点加运算和 2 倍点运算的执行次数。算法描述如下所示。

输入：椭圆曲线上的点 P 以及二进制整数 $k = \sum_{j=0}^{l-1} k_j 2^j, k_j \in \{0,1\}$，$l$ 表示最高位且为 1。

输出：椭圆曲线上的点 $Q = [k]P$。

第一步　令 $Q = P$；

第二步　for i from $l-1$ to 1 do

$Q = 2Q$；

若 $k_i = 1$，利用点加运算实现 $Q = Q + P$；

第三部　输出 Q。

为了计算上的方便，令最高位即第 l 位是 1。在该算法中，需要计算 $\log|k|$ 次 2 倍点运算和平均 $\frac{1}{2}\log(|k|)$ 次点的加法运算，因此共需要 $\frac{3}{2}k$ 次椭圆曲线的运算。

7.2.4　Hash 算法设计

SM2 加解密算法中所用到的 Hash 运算，采用的是我国商密算法 SM3。关于 SM3 算法的相关原理与实现，在本书中给出了详细的介绍，请参照本书第十章。另外，为了设计的方便，SM2 算法中涉及的 KDF（密钥派生函数）均用 SM3 算法代替。

7.2.5　模逆运算设计

通常素数域上的求逆算法有扩展的欧几里德算法和 Montgomery 求逆算法，扩展的欧几里德算法可以用来求两个数的最大公约数，也可以求模逆，算法的原理是辗转相除法，因此过程中用到了很多的除法和减法操作，不利于硬件的实现。

本书从降低算法层次和减少触发操作的角度出发，对其进行了改进，改进算法如下。

输入：模 p，整数 a，求 a 的逆元；

输出：$a^{-1} \bmod p$。

第一步　$(x,u) \leftarrow (1,a)$；

第二步　$(y,v) \leftarrow (0,p)$；

第三步　当 $u \neq 1$ 和 $v \neq 1$ 时，重复执行下面操作

当 $u_0 = 0$，重复执行 $u = u \gg 1$；

若 $x_0 = 0$，$x = x \gg 1$；

否则，$x = x + p \gg 1$；

当 $v_0 = 0$，重复执行 $v = v \gg 1$；

若 $y_0 = 0$，$y = y \gg 1$；

否则，$y = y + p \gg 1$

当 $u \geqslant v$，$(x,u) = (x - y, u - v)$；

否则，$(y,v) = (y - x, v - u)$；

当 $u = 1$，则返回 $x = a^{-1} \bmod p$，否则返回 $y = a^{-1} \bmod p$。

算法中使用公约数的性质，将扩展的欧几里德算法中的除法运算全部改成了减法，指令时间比除法操作要短；另外，将二进制中的除法运算运用到素数域里，并且经过改进能将素数域和二元域的求模逆运算结合起来，这样在硬件实现时，可以使用相同的控制单元，有利于节省成本。

7.3　SM2 算法工程实现

1．创建 ISE 工程

打开 ISE 开发环境，选择菜单 File→New Project，建立一个新工程，然后设置顶层文件模块名、存储目录和设计方式。接着选择器件，这里选择 Xilinx 的 Virtex4 来实现。最后再选择第三方仿真软件 ModelSim 和 Verilog 语言进行仿真测试。新工程的建立如图 7.4 和图 7.5 所示。

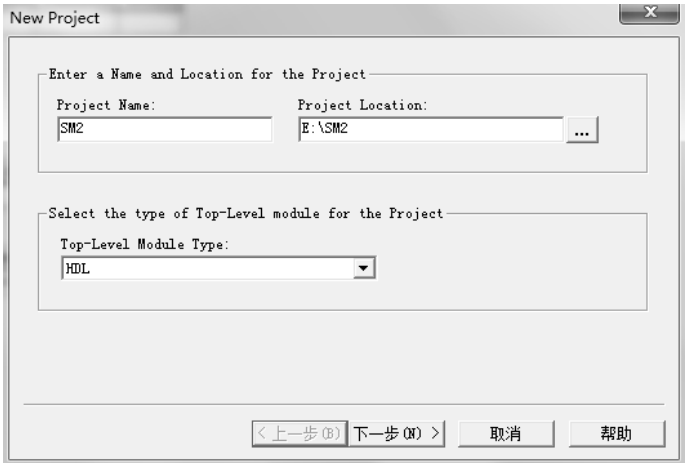

图 7.4 SM2 算法工程建立

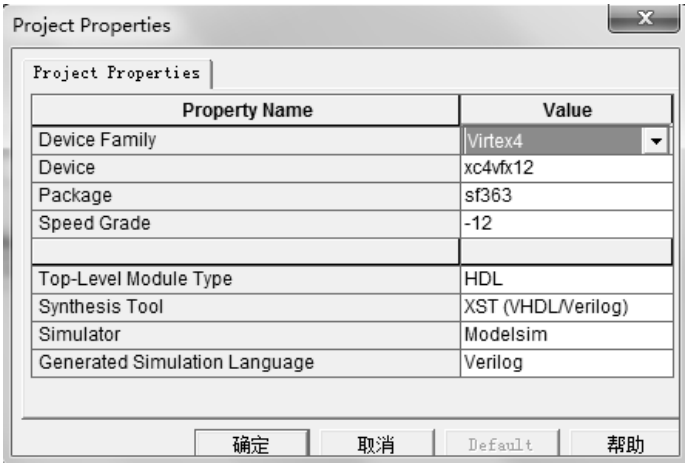

图 7.5 器件与编程语言的选择

2．编写设计代码

在 7.2 节中详细叙述了 SM2 算法的关键模块设计，上面的模块就不再赘述。以下主要讨论模乘算法的设计与实现。模乘算法的参考程序如程序 7-1 所示。

程序 7-1：

```
…
reg   first;
reg [MPWID-1:0] mpreg;
reg [MPWID+1:0] prodreg;
reg [MPWID+1:0] mcreg;
reg [MPWID+1:0] modreg1,modreg2;
wire[MPWID+1:0] mcreg1,mcreg2;
wire reset;
wire [MPWID-1:0] mpand;
```

```
 wire [MPWID-1:0] mplier;
 wire [MOD_LENGTH-1:0] modulus;
 wire  [1:0] modstate;
 wire [MPWID+1:0] prodreg1,prodreg2,prodreg3,prodreg4;
 assign prodreg1 = (mpreg[0]==1'b1)?(prodreg+mcreg):prodreg;
 assign prodreg2 = prodreg1- modreg1;
 assign prodreg3 = prodreg1 - modreg2;
 assign modstate = {prodreg3[MPWID+1],prodreg2[MPWID+1]};
 assign prodreg4 =(modstate==2'b11)? prodreg1:(modstate==2'b10)?
                  prodreg2:prodreg3;
 assign mcreg1 = mcreg - modreg1;
 assign mcreg2 = mcreg1[MPWID]? mcreg:mcreg1;
 assign ready = first;
 always@(posedge ready)
 begin
   product = prodreg4[MPWID-1:0];
   end
 always@( posedge clk  or negedge reset)
 begin
if (!reset)
 begin
  first = 1'b1;
  end
else
 begin
  if (first)
      begin
             if(ds)
             begin
                 mpreg = mplier;
                 mcreg ={2'b00,mpand};
                 modreg1={2'b00,modulus};
                 modreg2={1'b0,modulus,1'b0};
                 prodreg = 0;
                 first = 0;
             end
         end
  else
     begin
         if(mpreg == 0)
             begin
                 first = 1'b1;
                 end
         else
```

```
                    begin
                        mcreg = {mcreg2[MPWID:0],1'b0};
                        mpreg = {1'b0,mpreg[MPWID-1 :1]};
                        prodreg = prodreg4;
                        end
            end
        end
        end
      endmodule
```

上述程序是使用 Montgomery 模乘算法来实现的，用到了三目运算符，这样简化了程序，并增加了程序的可读性。

将以上实现的模块代码导入新建的工程中，具体如图 7.6 所示。

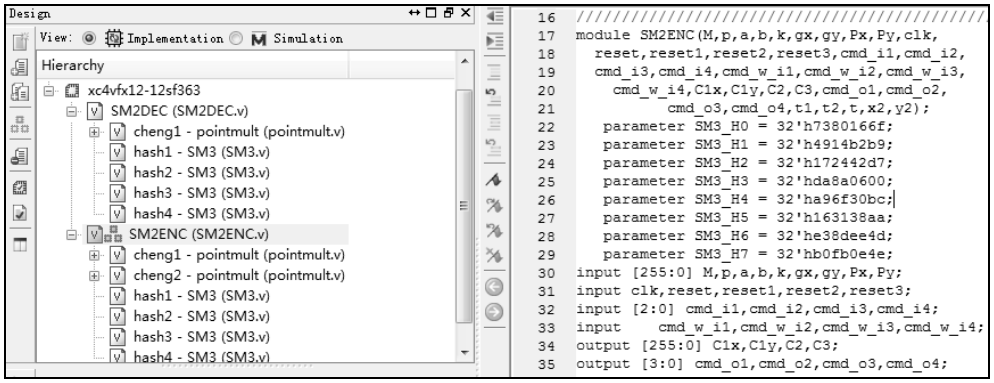

图 7.6 SM2 算法的实现

3．工程验证

（1）根据以上编写的程序，将以上模块加入 ISE 工程文件，并编写该程序的测试文件。如图 7.7 和图 7.8 所示。

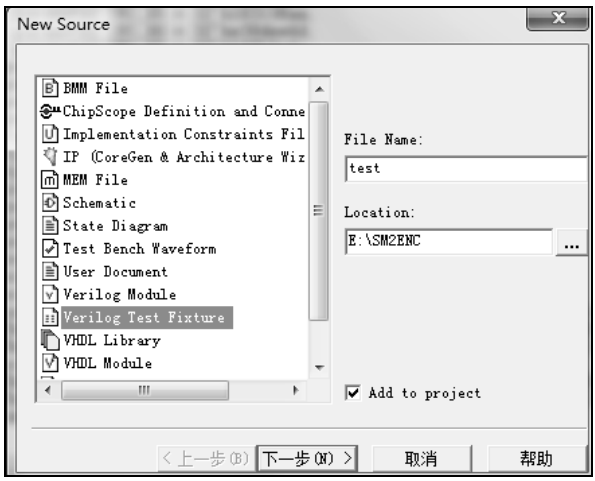

图 7.7 SM2 算法测试文件的建立

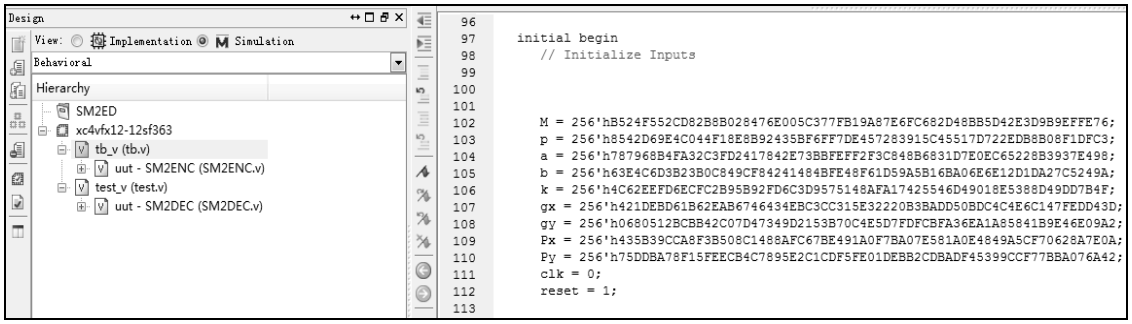

图 7.8　SM2 算法测试文件

（2）根据上一步的测试文件，调用 ModelSim 仿真软件，如图 7.9 所示，对该算法进行仿真测试。仿真结果如图 7.10 和图 7.11 所示。

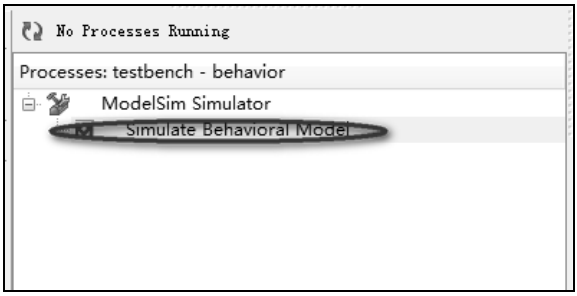

图 7.9　调用 ModelSim 仿真

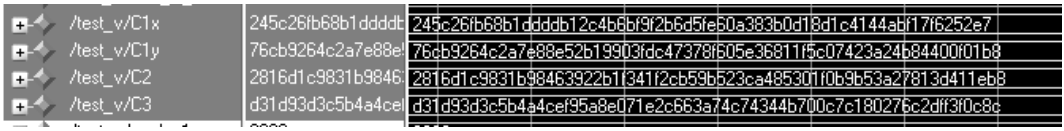

图 7.10　SM2 算法加密结果

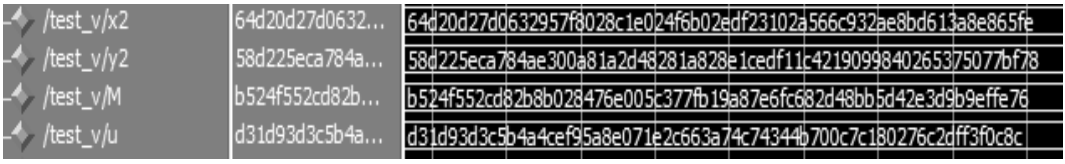

图 7.11　SM2 算法解密结果

7.4　效果测试

通过功能仿真与系统测试，可以验证整个设计的正确性。对此仿真时，在波形文件中需要给出合适的时钟周期。在 FPGA 程序设计中，验证算法的正确性和设计合理性的方法之一是对程序进行波形仿真。在本书中使用 ModelSim 软件对算法进行仿真，以验证其功能是否正确。下面是 SM2 算法加解密过程中的测试数据。

加密过程。

明文：B524F552CD82B8B028476E005C377FB19A87E6FC682D48BB5D42E3D9B9EFFE76；

模数：8542D69E4C044F18E8B92435BF6FF7DE457283915C45517D722EDB8B08F1DFC3；

参数 a：787968B4FA32C3FD2417842E73BBFEFF2F3C848B6831D7E0EC65228B3937E498；

参数 b：63E4C6D3B23B0C849CF84241484BFE48F61D59A5B16BA06E6E12D1DA27C5249；

倍数 k：4C62EEFD6ECFC2B95B92FD6C3D9575148AFA17425546D49018E5388D49DD7B4；

基点：（421DEBD61B62EAB6746434EBC3CC315E32220B3BADD50BDC4C4E6C147FEDD43D，0680512BCBB42C07D47349D2153B70C4E5D7FDFCBFA36EA1A85841B9E46E09A2）；

公钥：（435B39CCA8F3B508C1488AFC67BE491A0F7BA07E581A0E4849A5CF70628A7E0A，75DDBA78F15FEECB4C7895E2C1CDF5FE01DEBB2CDBADF45399CCF77BBA076A）；

密文：C1=（245c26fb68b1ddddb12c4b6bf9f2b6d5fe60a383b0d18d1c4144abf17f6252e7，76cb9264c2a7e88e52b19903fdc47378f605e36811f5c07423a24b84400f01b8）；

C2=2816d1c9831b98463922b1f341f2cb59b523ca485301f0b9b53a27813d411eb8；

C3=d31d93d3c5b4a4cef95a8e071e2c663a74c74344b700c7c180276c2dff3f0c8c。

解密过程。

私钥：1649AB77A00637BD5E2EFE283FBF353534AA7F7CB89463F208DDBC2920BB0DA0；

明文：C1=（245c26fb68b1ddddb12c4b6bf9f2b6d5fe60a383b0d18d1c4144abf17f6252e7，76cb9264c2a7e88e52b19903fdc47378f605e36811f5c07423a24b84400f01b8）；

C2=2816d1c9831b98463922b1f341f2cb59b523ca485301f0b9b53a27813d411eb8；

C3=d31d93d3c5b4a4cef95a8e071e2c663a74c74344b700c7c180276c2dff3f0c8c；

密文：B524F552CD82B8B028476E005C377FB19A87E6FC682D48BB5D42E3D9B9EFFE76。

下面对该工程的时序、运行效率、资源占用情况进行分析。在 Xilinx 公司的 6slx150fgg484-3 FPGA 元器件上，使用 ISE 的 XST 分析工具进行综合。综合结果如图 7.12 所示。

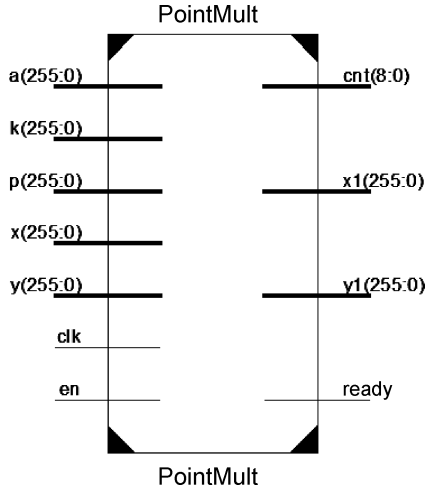

图 7.12　SM2 算法点乘运算综合后的外部器件

点乘运算资源占用情况分析报告如表 7.1 所示。

表 7.1　SM2 算法点乘运算资源占用情况

Selected Device : 6slx150fgg484-3	
Number of Slices	21　out of　184304　0%
Number of Slice Flip Flops	92　out of　92152　0%
Number of 4 input LUTs	92　out of　92152　0%
Number of IOs	96
Number of bonded IOBs	75　out of　96　78%

时序分析报告如下。

Minimum period: 5.040ns (Maximum Frequency: 198.419MHz)

Minimum input arrival time before clock: 4.907ns

Maximum output required time after clock: 4.151ns

Maximum combinational path delay: No path found

综合分析可知，影响 SM2 算法加解密效率的决定因素是点乘运算的速度。根据实际的仿真测试，得到点乘运算速度为：260 次/秒，可以满足实际需要。

7.5 本章小结

SM2 算法是我国自主知识产权的商用公钥密码算法，是 ECC 算法的一种。

本章首先对 SM2 算法的原理、工作模式与加解密流程进行了介绍，之后重点介绍了 SM2 算法的 FPGA 实现方法，重点包括点加运算、2 倍点运算、多倍点运算和模逆运算的设计，并给出了点乘运算的硬件实现方案，最后给出了仿真结果并对结果进行了分析，使读者不仅能从理论上了解 SM2 算法，更能从算法的 FPGA 实现过程中加深对算法的理解和掌握。

第 8 章　SHA-1 算法 FPGA 实现

SHA-1 算法由美国国家标准技术研究院（NIST）与美国国家安全局（NSA）设计，并且被美国政府采纳，成为美国国家标准。事实上 SHA-1 算法目前是全世界使用最为广泛的 Hash 算法之一，SHA-1 算法可以对长度不超过 2^{64} 比特的消息进行计算，产生 160 比特的消息摘要作为输出。

8.1　SHA-1 算法原理

8.1.1　SHA-1 算法的补位与补长度

在 SHA-1 算法中，必须把原始消息（字符串，文件等）转换成位字符串，SHA-1 算法只接受位作为输入。假设对字符串“*abc*”产生消息摘要，首先，将它转换成位字符串如下：

```
01100001 01100010 01100011
--------------------------
'a'=97   'b'=98    'c'=99
```

这个位字符串的长度为 24。

消息需要进行补位，以使其长度在对 512 取模以后的余数是 448，也就是说，（补位后的消息长度）mod 512 = 448。即使长度已经满足对 512 取模后余数是 448，补位也要进行。补位是这样进行的：先补一个 1，然后再补 0，直到长度满足对 512 取模后余数是 448。总而言之，补位是至少补 1 位，最多补 512 位。还是以前面的“*abc*”为例展示补位的过程。原始信息： 01100001 01100010 01100011

补位第一步：补一个“1”　　01100001 01100010 01100011 1

补位第二步：补 423 个“0”　　01100001 01100010 01100011 10…0

可以把最后补位完成后的数据用 16 进制表示如下：

61626380 00000000 00000000 00000000
00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000
00000000 00000000

所谓的补长度是将原始数据的长度补到已经进行了补位操作的消息后面。通常用一个 64 位的数据来表示原始消息的长度，如果消息长度不大于 2^{64} 位，那么第一个字就是 0。在进行了补长度的操作以后，整个消息就变成如下所示（16 进制格式，共 512 位）：

61626380 00000000 00000000 00000000
00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000018

如果原始的消息长度超过了 512 位，需要将它补成 512 位的倍数。然后把整个消息分成一个个 512 位的数据块，分别处理每一个数据块，从而得到消息摘要。

8.1.2　计算消息摘要

必须使用进行了补位和补长度后的消息来计算消息摘要。计算需要两个缓冲区，每个都由 5 个 32 位的字组成，还需要一个能容纳 80 个 32 位字的加大缓冲区。第一个 5 个字的缓冲区被标识为 A,B,C,D,E，第二个 5 个字的缓冲区被标识为 H_0,H_1,H_2,H_3,H_4，80 个字的缓冲区被标识为 $W_0,W_1,\cdots,W_{79}$，另外还需要一个 1 个字的 TEMP 缓冲区。

为了产生消息摘要，定义的 16 个字（每个字 32bit）的数据块 $M_1,M_2,\cdots,M_n$ 会依次进行处理，处理每个数据块 M_i 要包含 80 个步骤。

在处理每个数据块之前，缓冲区 $\{H_i\}$ 被初始化为下面的值（十六进制）：

$$H_0 = 0\text{x}67452301$$
$$H_1 = 0\text{xefcdab89}$$
$$H_2 = 0\text{x98badcfe}$$
$$H_3 = 0\text{x10325476}$$
$$H_4 = 0\text{xc3d2e1f0}$$

现在开始处理 $M_1,M_2,\cdots,M_n$，为了处理 M_i，需要进行下面的步骤。

（1）将 M_i 分成 16 个字 $W_0,W_1,\cdots,W_{15}$，其中 W_0 是最左边的字；

（2）对于 $t=16$ 到 79，令 $W_t = S^1(W_{t-3}\ \text{XOR}\ W_{t-8}\ \text{XOR}\ W_{t-14}\ \text{XOR}\ W_{t-16})$；

（3）令 $A=H_0$，$B=H_1$，$C=H_2$，$D=H_3$，$E=H_4$；

（4）对于 $t=0$ 到 79，执行循环：$\text{TEMP} = S^5(A) + f(t,B,C,D) + E + W_t + K_t$；

$E=D$；$D=C$；$C=S^{30}(B)$；$B=A$；$A=\text{TEMP}$；

（5）令 $H_0 = H_0 + A$，$H_1 = H_1 + B$，$H_2 = H_2 + CH_3 = H_3 + D$，$H_4 = H_4 + E$。

式中，S 表示循环左移，上标表示左移位数。

在处理完所有的 M_n 后，消息摘要是一个 160 位的字符串，以顺序标识为 $H_0H_1H_2H_3H_4$，对于 SHA256，SHA384，SHA512，也可以用相似的办法来计算消息摘要，对消息进行补位的算法完全是一样的。

8.2　SHA-1 算法基本步骤

步骤 1：填充消息。在消息比特流之后，先填上一个 1，后跟一串 0，使得消息的长度比 512 的倍数少 64 位，最后再加上一个 64 位的对填充之前消息长度的表示。这样使消息长度恰好是 512 的倍数，同时保证不同的消息在填充后仍然不同。

步骤 2：变量初始化。要使得 SHA-1 的输出为 160 位，需要对 5 个 32 位变量初始化，则 $A=0\text{x}67452301$，$B=0\text{xefcdab89}$，$C=0\text{x98badcfe}$，$D=0\text{x10325476}$，$E=0\text{xc3d2elf0}$。

步骤 3：以 512 位数据块为单位处理消息。算法的核心是一个包含 4 个循环的模块，每个循环由 20 个处理步骤组成。4 个基本逻辑函数为 f_1、f_2、f_3、f_4，每个循环使用不同的逻辑函数。函数定义如下。

$$f_1 = f(t,B,C,D) = (B \wedge C) \vee (\bar{B} \wedge D)(0 \leqslant t \leqslant 19)$$
$$f_2 = f(t,B,C,D) = B \oplus C \oplus D(20 \leqslant t \leqslant 39)$$
$$f_3 = f(t,B,C,D) = (B \wedge C) \vee (B \wedge D) \vee (C \wedge D)(40 \leqslant t \leqslant 59)$$
$$f_4 = f(t,B,C,D) = B \oplus C \oplus D(60 \leqslant t \leqslant 79)$$

其中，“$\wedge$”是与操作，“$\vee$”是或操作，“$\bar{B}$”表示对 B 取反操作，“$\oplus$”是异或操作。

每个循环都以当前正在处理的 512 位和 160 位的缓存值 A，B，C，D，E 为输入，然后更新缓存内容。每个循环还使用一个 32 位常数值 K_t，这些值的定义如表 8.1 所示。

表 8.1 不同步骤使用的 K_t 值

步 数	十六进制表示
$0 \leqslant t \leqslant 19$	$K_t = 0\text{x}5\text{a}827999$
$20 \leqslant t \leqslant 39$	$K_t = 0\text{x}6\text{ed}9\text{eba}1$
$40 \leqslant t \leqslant 59$	$K_t = 0\text{x}8\text{f}1\text{bbcdc}$
$60 \leqslant t \leqslant 79$	$K_t = 0\text{xca}62\text{c}1\text{d}6$

步骤 4：结果输出。全部 512 位数据块处理完毕后，最后输出的就是 160 位消息摘要。

SHA1 散列函数如图 8.1 所示（式中 S 表示循环左移，上标表示移位位数），它是处理每个 512 位分组的所有循环的逻辑运算。循环的变换形式为

$$(A,B,C,D,E) = ((E + f(t,B,C,D) + S^5(A) + W_t + K_t), A, S^{30}(B), C, D)$$

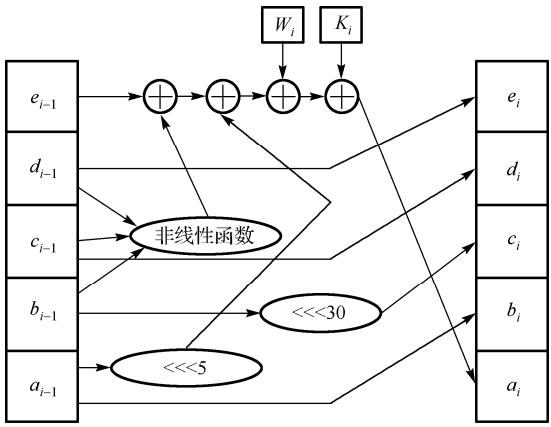

图 8.1 SHA-1 压缩函数中的逻辑运算

W_t 的定义如下：

$W_t = M_t$，$t = 0 \sim 15$；

$W_t = (W_{t-3} \oplus W_{t-8} \oplus W_{t-14} \oplus W_{t-16}) <<< 1$，对于 $t = 16 \sim 79$；

8.3 SHA-1 算法的 FPGA 设计

SHA-1 算法进行 FPGA 实现时，整体结构可分为四个模块：控制单元模块、消息扩展模块、迭代压缩模块、结果输出模块。其关键模块如图 8.2 所示。

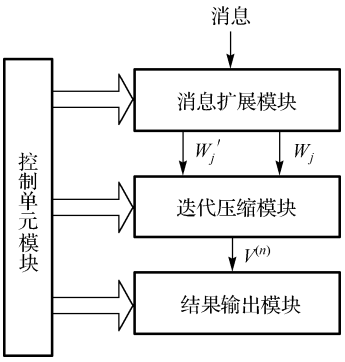

图 8.2　SHA-1 算法关键模块

8.3.1　控制单元模块设计

控制信号比较简单且有规律，所以在本设计中采用了计数器控制方式。由于每处理 512bit 明文用要 81 个时钟周期，所以计数器采用 7 位，总共有 81 个状态，通过对 state、start、rnd_cnt_q 的设置来控制这 81 个状态，从而控制消息扩展模块、迭代压缩模块和结果输出模块工作。

控制单元模块的程序实现，参考如程序 8-1 所示。

程序 8-1：

```
always @(state or start or rnd_cnt_q)
begin
   out_valid = 1'b0;
   busy = 1'b0;
   rnd_cnt_d = 7'b0000000;
   case (state)
     IDLE : begin
        out_valid = 1'b0;
        rnd_cnt_d = 7'b0000000;
        if (start) begin
           busy = 1'b1;
           next_state = CALC;
        end
        else begin
           busy = 1'b1;
           next_state = IDLE;
        end
     end // case: IDLE
     CALC : begin
        busy = 1'b1;
        out_valid = 1'b0;
        if (start) begin
           next_state = IDLE;
           rnd_cnt_d = 7'd0000000;
        end
        else if (rnd_cnt_q == 7'd79) begin
           next_state = VALID_OUT;
```

```
            rnd_cnt_d = 7'd0000000;
          end
          else begin
            next_state = CALC;
            rnd_cnt_d = rnd_cnt_q + 7'd0000001;
          end
        end // case: CALC
```

上述控制单元模块的程序代码中，可以看出，利用 case 语句，通过对 rnd_cnt_q 和 start 的控制，得到 next_state 状态；通过对 busy 的控制，得到 out_valid，结果存放在 rnd_cnt_d 中。

8.3.2 消息扩展模块设计

每个生成的W_j在轮运算模块中只用到一次，本设计采用 16 个 32 位的寄存器（$m_0 \cdots m_f$）来存储每一步中可能用到的 16×32bit 数据，每一步的W_j使用完以后便让出空间，而不是用一个 80×16×32bit 的存储器来存储 80 个W_j，因此节约了大量资源，同时也提高了速度。

消息扩展模块的程序实现，参考程序 8-2。

程序 8-2：

```
wire [31:0]     w_gen_temp = w_temp[511:480] ^ w_temp[447:416]
                    ^ w_temp[255:224] ^ w_temp[95:64];
wire [31:0]     w_gen = {w_gen_temp[30:0], w_gen_temp[31]};
                //选择即将输入的 msg 值或者计算出来的值
assignw_d = ((load_in == 1'b1) ? w_temp : ((rnd_cnt_q< 7'd15) ?w_temp :
                        {w_gen, w_temp[479:0]}));   // msg 的值左移
                                                    //左循环移位 1 个字节
assignw_temp = (state == CALC) ? {w_q[479:0], w_q[511:480]} : ((load_in
                        == 1'b1) ? {w_q[479:0], data_in} :w_q)
                // w 的输入直接由累加寄存器的输出得到
assign w = w_q[511:480];
sha1_round sha1_round(.cv_in(rnd_q), .w(w), .round(rnd_cnt_q),
                      .cv_out(sha1_round_wire));
```

wire 语句是选择即将输入的 msg 值或者计算出来的值。assign 的作用是作为信号量输出，通过寄存器连续赋值；也可以作为信号量输出，通过寄存器拼接数据位实现；还可以作为信号量输出，通过判断条件，赋值给信号。

8.3.3 迭代压缩模块设计

该运算一共四轮，每轮运算的框架大体相同，只是常数和逻辑函数不同，所以四轮是共用同一模块。压缩模块是系统的关键模块，是 SHA-1 算法的关键步骤，是硬件实现不可或缺的环节。迭代压缩模块的程序实现，参考程序 8-3。

程序 8-3：

```
module sha1_round (cv_in, w, round, cv_out);
input [159:0] cv_in;
input [31:0]   w;
```

```
input [6:0]    round;
output [159:0] cv_out;
reg [31:0]     k;
reg [31:0]     f;
wire [31:0]    a_shift;
wire [31:0]    b_shift;
wire [31:0]    add_result;
wire [31:0]    a = cv_in[159:128];
wire [31:0]    b = cv_in[127:96];
wire [31:0]    c = cv_in[95:64];
wire [31:0]    d = cv_in[63:32];
wire [31:0]    e = cv_in[31:0];
   //建立 4 个类似的比较器
always @(round)
begin
        k = 32'd0;
if ((round >= 7'd0) && (round <= 7'd19))
        k = 32'h5A827999;
if ((round >= 7'd20) && (round <= 7'd39))
        k = 32'h6ED9EBA1;
if ((round >= 7'd40) && (round <= 7'd59))
        k = 32'h8F1BBCDC;
if ((round >= 7'd60) && (round <= 7'd79))
        k = 32'hCA62C1D6;
end // always @ (round)
always @(round or b or c or d)
begin
        f = 32'd0;
if ((round >= 7'd0) && (round <= 7'd19))
        f = ((b & c) | (~b & d));
if ((round >= 7'd20) && (round <= 7'd39))
        f = (b ^ c ^ d);
if ((round >= 7'd40) && (round <= 7'd59))
        f = ((b & c) | (b & d) | (c & d));
if ((round >= 7'd60) && (round <= 7'd79))
        f = (b ^ c ^ d);
end // always @ (round or b or c or d)
assigna_shift = {a[26:0], a[31:27]};
assignb_shift = {b[1:0], b[31:2]};
   // e 和 w 来自寄存器的输出
   // k 是一个 6bit 的比较器或者多路复用器
   // f 是一个 6bit 的比较器或者多路复用器或者计算器
   // a 是之前的循环左移 5 位得到的
assignadd_result = (a_shift + ((f + k) + (e + w)));
assigncv_out = {add_result, a, b_shift, c, d};
endmodule // sha1_round
```

wire 表示直通，即只要输入有变化，输出马上无条件地反映出来；reg 表示一定要有触发，输出才会反映输入。代码有四轮迭代，每一轮都是一样的，所以只需要写一轮就可以了。begin 语段的分支结构为：如果（round >= 7'd0) && (round <= 7'd19)，则 $f = ((b \& c) | (\sim b \& d))$；如果(round >= 7'd20) && (round <= 7'd39)，则 $f = (b \wedge c \wedge d)$；如果(round >= 7'd40) && (round <= 7'd59)，则 $f = ((b \& c) | (b \& d) | (c \& d))$；如果(round >= 7'd60) && (round <= 7'd79)，则 $f = (b \wedge c \wedge d)$；由此得到不同步骤使用的 K_t 值，其中，e 和 w 是来自于寄存器的输出，k 有 6bit 的多路复用器的延迟，a 从之前的循环中左移 5 位得到。

8.3.4 结果输出模块设计

根据控制电路给出的状态消息输出模块，按照算法要求的步骤即可完成消息的输出。具体过程如下：状态机通过对 rnd_cnt_q 和 state 的控制，从而分四段完成对 Hash 值的输出。本实例设计的数据端口为 32bit 宽，为了便于将设计模块挂载到当前常用的数据总线上，数据按照 4 个周期 32bit 进入 Hash 单元。结果输出模块的程序实现，参考程序 8-4。

程序 8-4：

```
    VALID_OUT : begin
                busy = 1'b1;
                out_valid = 1'b0;
                // Allow cycle to latch output
                if (start) begin
                   next_state = IDLE;
                   rnd_cnt_d = 7'd0000000;
                end else begin
                   next_state = VALID_OUT2;
                end
              end
              VALID_OUT2 : begin
                busy = 1'b0;
                out_valid = 1'b1;
                if (start) begin
                   next_state = CALC;
                end
                else begin
                   next_state = IDLE;
                end
              end
              default : begin
                next_state = IDLE;
              end
            endcase
     end
```

在以上代码中，在第 80 个状态循环，得到最终结果，利用 if 语句，通过对 start、busy 和 out_valid 的设置，控制 if 的选择条件，得到 Hash 值，最终的结果存放在 rnd_cnt_d 中。

8.4 SHA-1 算法工程实现

1．创建 ISE 工程

打开 ISE 开发环境，选择菜单 File→New Project，建立一个新工程，然后设置顶层文件模块名、存储目录和设计方式。接着选择器件，这里选择 Xilinx 公司的 Virtex4 来实现。最后再选择第三方仿真软件 ModelSim 和 Verilog 语言进行仿真测试。新工程的建立如图 8.3 和图 8.4 所示。

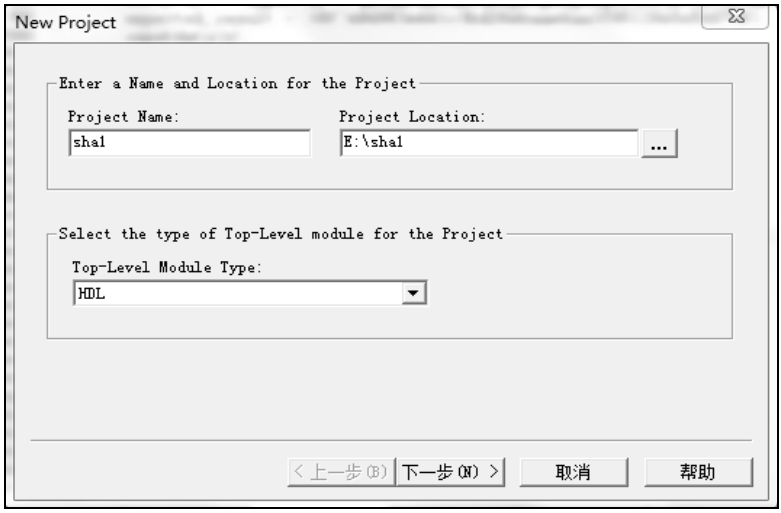

图 8.3　SHA-1 算法工程文件的建立

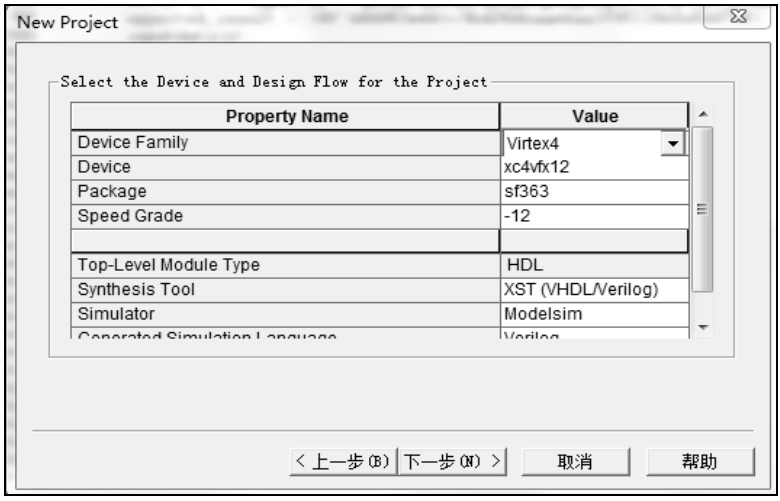

图 8.4　器件的选择和工具的使用

2．编写设计代码

在 8.3 节中详细叙述了 SHA-1 算法的关键模块设计，上述模块在 ISE 中的具体实现如图 8.5 和图 8.6 所示。

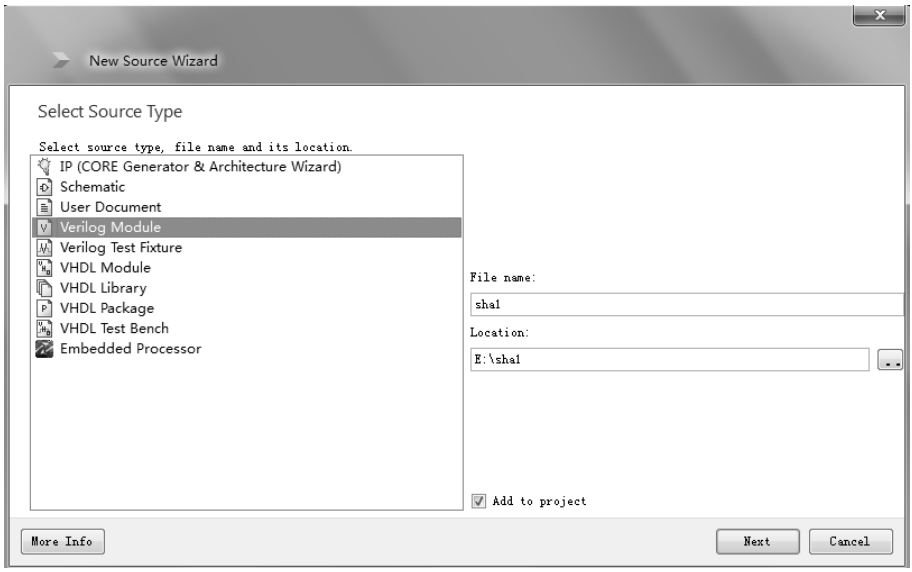

图 8.5　新建工程

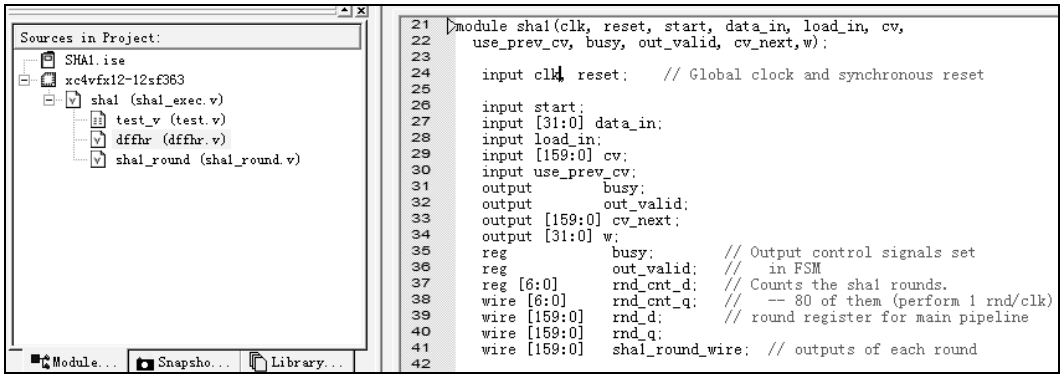

图 8.6　打开工程

3．工程验证

（1）根据以上编写的程序，编写该程序的测试文件，如图 8.7 所示。

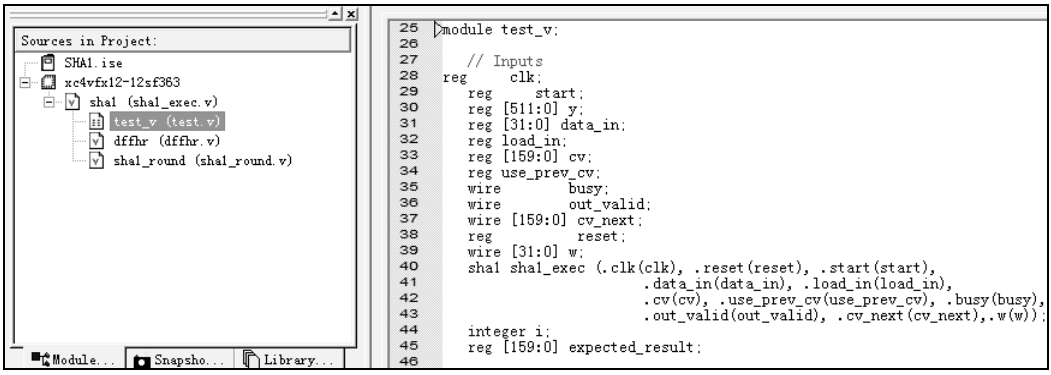

图 8.7　SHA-1 算法测试文件

（2）根据上一步的测试文件，利用 ModelSim 仿真软件，如图 8.8 所示，对该算法进行仿真测试。仿真结果如图 8.9 所示。

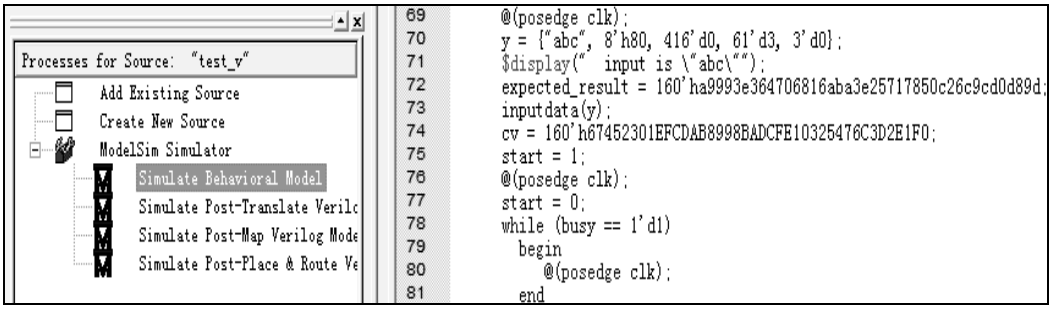

图 8.8　ModelSim 调用示意图

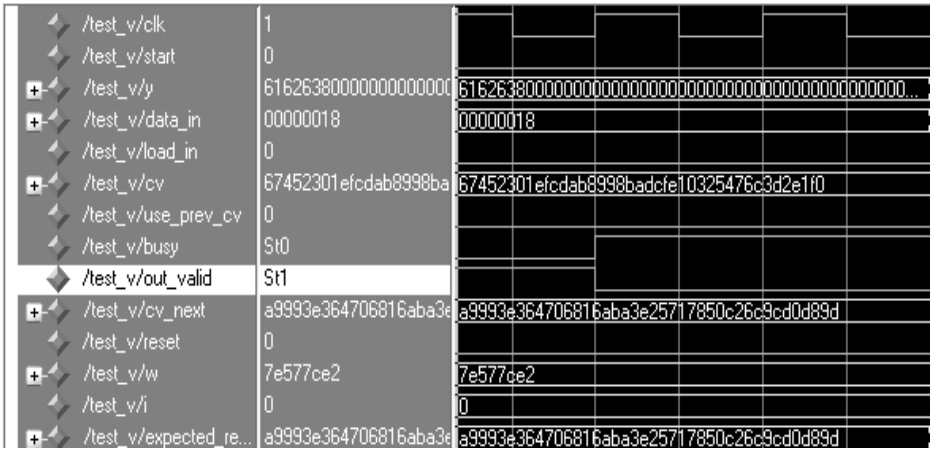

图 8.9　“*abc*”经过 SHA-1 算法运算后的仿真测试结果

8.5　效果测试

由以上的仿真与系统测试，可以验证整个设计的正确性，为了测试其硬件功能，还需要对整个设计在芯片上的综合结果进行测试分析。

在 FPGA 程序设计中，对程序进行波形仿真是验证算法的正确性与合理性的有效方法。本实例使用 ModelSim 对算法进行仿真，以验证算法运算是否正确。例如：使用 16 进制的明文 *abc*，其 SHA-1 运算的结果为 a9993e36 4706816a ba3e2571 7850c26c 9cd0d89d。下面测试比较长的明文，如测试（16 进制）abcdbcdecdefdefgefghfghighijhijkijkljklmklmnlmnomnopnopq，其 SHA-1 运算的结果为 84983e44 1c3bd26e baae4aa1 f95129e5 e54670f1。这两个实例测试与通过其他途径获得的 Hash 值是一致的，也就是说，本实例的 SHA-1 算法 FPGA 设计的正确性得到了验证。仿真的波形如图 8.10 所示。

下面对该工程的时序、运行效率、资源占用情况进行分析。在 Xilinx 公司的 Virtex4 FPGA 元器件，经过 ISE 的 XST 分析工具进行综合，综合结果如图 8.11 所示。

资源占用情况分析报告如表 8.2 所示。

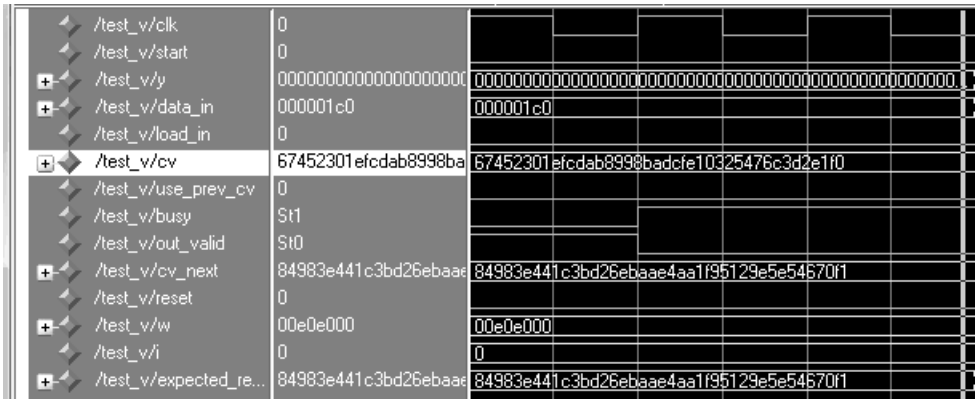

图 8.10　较长明文经过 SHA-1 运算后的测试结果

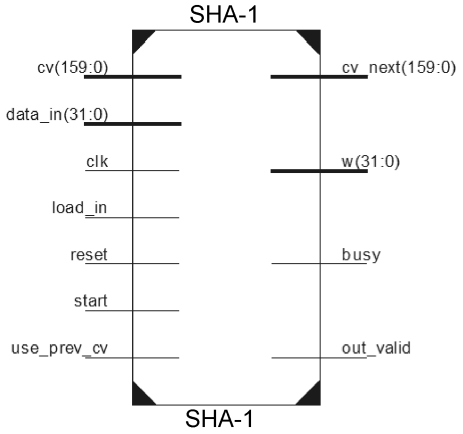

图 8.11　SHA-1 算法综合后的外部器件

表 8.2　SHA-1 算法资源占用情况

Selected Device : 4vfx12sf363-12	
Number of Slices	850　out of　12480　　6%
Number of Slice Flip Flops	1005　out of　12480　　8%
Number of 4 input LUTs	941　out of　12480　　7%
Number of IOs	391
Number of bonded IOBs	391　out of　　172　　227% (*)
Number of BUFG/BUFGCTRLs	1　out of　　32　　3%

时序分析报告如下。

Timing Summary:

Speed Grade: -2

Minimum period: 7.252 ns (Maximum Frequency: 137.898 MHz)

Minimum input arrival time before clock: 3.103 ns

Maximum output required time after clock: 3.559 ns

Maximum combinational path delay: No path found

由时序分析报告可知：SHA-1 算法的最高工作频率为 137.898 MHz。分析程序可知，每

个 512 比特的分组需要经过 102 个时钟，所以，SHA-1 算法的最高处理速度为 512×137.898/102= 692.19 Mbps。

由以上分析报告，可以看出，整个工程在 Xilinx 公司的 4vfx12 芯片上使用的查找表数为 941，占芯片资源的 7%。虽然占用引脚资源比较多，有待改进，但速度很快，满足了运算效率的要求。

8.6 本章小结

本章利用 FPGA 并行处理技术实现了 SHA-1 算法，并通过综合仿真验证了设计的正确性。此设计可以应用于多种场合，将此设计集成到其他 FPGA 设计中，可以实现对消息的加密、签名、认证及密钥交换。通过本章的学习，能比较深刻地理解 SHA-1 算法的原理与实现步骤，掌握用 FPGA 实现 SHA-1 算法所使用的模块，通过仿真更为直观地了解此算法代码的工作效果。

第 9 章　Keccak 算法 FPGA 实现

由于近年来对传统常用 Hash 函数（如 MD4、MD5、SHA-1、RIPENMD 等）的成功攻击，美国国家标准技术研究所（NIST）分别在 2005 年、2006 年举行了两届密码 Hash 研讨会；于 2007 年正式宣布在全球范围内征集新的下一代密码 Hash 算法，举行 SHA-3 竞赛。新的 Hash 算法将被称为 SHA-3，并且作为新的安全 Hash 标准，增强现有的 FIPS 180-2 标准。候选算法提交于 2008 年 10 月结束，NIST 分别于 2009 年和 2010 年举行了 2 轮会议，通过 2 轮的筛选选出了进入最终轮（Final round）的 5 个 Hash 算法（BLAKE、Grφstl、JH、Keccak、Skein）。2012 年 10 月，NIST 宣布 Keccak 算法为新的 Hash 标准算法。

Keccak 是由 ST 微电子（ST Microelectronics）公司的 Bertoni、Daemen、Assche 和 NXP 半导体公司的 Peeters 共同设计提交的 Hash 算法，它使用基于 Sponge 构造的 Sponge 函数。Keccak 的状态更新使用的是作用在三维数组上的 5 步迭代排列。

9.1　算法描述

9.1.1　Keccak 结构

Keccak 是一种采用密封海绵结构（Hermetic Sponge Strategy，HSS）的安全散列算法，由置换函数 Keccak-f 和特定的填充函数组成，如图 9.1 所示。其中，P 为输入，Z 为 Hash 后的输出，f 为置换函数，r 为波特率，c 为容量。输入数据通过填充函数进入 state，然后在 state 中被置换函数 f 处理后输出。state 可看成一个 5×5×w 的三维数组，用 a[5][5][w]表示，如图 9.2 所示，lane 的长度为 w，可以看作是 w 位 CPU 的 1 个字节。

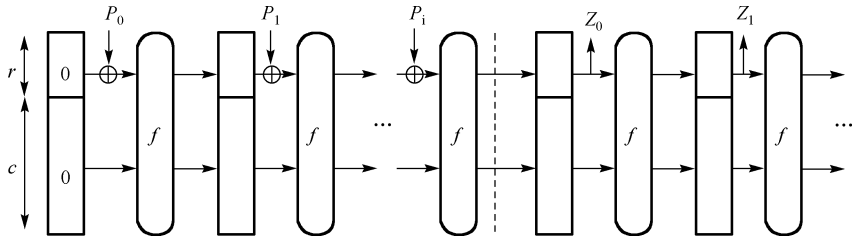

图 9.1　Keccak 海绵结构

Keccak-f 置换共有 7 种，用 Keccak-f[b]表示，其中 b 为置换宽度 $b\in\{25，50，100，200，400，800，1600\}$，$b$=25$w$，参加 SHA-3 竞赛的 Keccak 统一采用 Keccak-f[1600]置换，结构如图 9.3 所示。通过不同的 r，c 组合实现不同版本参数的 Keccak 函数，它们的不同组合会对安全和性能产生影响，要求 $c\geqslant 2n$；因为 c 和 r 的组合不同，导致生成的 Hash 值也不同，而不是简单地减少了输出的长度，这是 Keccak 能够进入到第 3 轮很重要的一个原因。

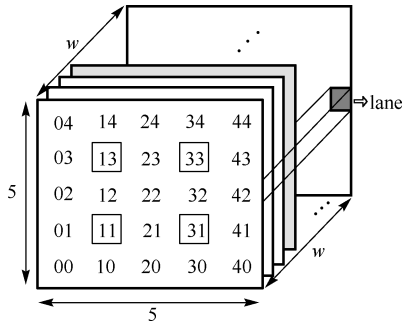

图 9.2　Keccak 的 state 结构

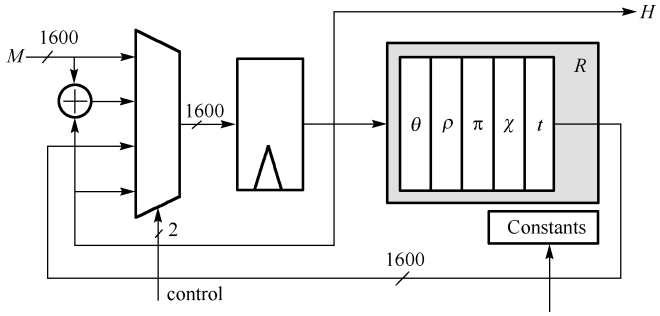

图 9.3　Keccak-f[1600]结构

图 9.3 中，输入数据经过填充处理封装成 *r* 位的数据，与 state 的初值（默认为 0）进行异或后封装成 1600 位的消息进入到 Keccak-f［1600］，再经过 R 循环迭代执行后输出 Hash 值，其中迭代次数 *nr* 由置换宽度 *b* 决定，它们之间的约束关系为 $nr=12+2l$，其中 $2l=b/25$。Keccak-f［1600］的 *R* 循环将执行 24 次，每个 *R* 循环只有最后一步的循环常数值不同。根据版本要求，从经过迭代置换后的 state 值中取出相应长度的信息摘要，即 Hash 值。

9.1.2　常数与函数

Keccak 使用基于 Sponge 构造的 Sponge 函数，Sponge 函数表示为 Keccak [*r*, *c*, *d*]。*r*、*c*、*d* 表示三个输入参数。在函数中，||表示连接符号，⊕表示按位异或，$\lfloor s \rfloor_n$ 表示将字符串 *s* 截短只留前面 *n* 比特。

Keccak [*r*, *c*, *d*]函数表示如下：

$$\text{Input}\quad M \in Z_2^*$$

$$\text{Output}\quad Z \in Z_2^*$$

$$P = \text{pad}(M,8) \,||\, \text{enc}(d,8) \,||\, \text{enc}(r/8,8)$$

$$P = \text{pad}(P,r)$$

$$\text{Let}\quad P = P_0 \,||\, P_1 \,||\, \cdots \,||\, P_{|P|-1}\,\frac{1}{2}\,,\text{with } P_i \in Z_2^r$$

$$s = 0^{r+c}$$

$$\text{For}\quad i = 0 \,\text{to}\, |P| - 1$$

{

$$s = s \oplus (P_i \| 0^c)$$
$$s = \text{Keccak} - f[r+c](s)$$

}

$Z = \text{empty string}$

While out is requested do

{

$$Z = Z \| \lfloor s \rfloor_r$$
$$s = \text{Keccak} - f[r+c](s)$$

}

Keccak – f 是作用在三维数组上的 5 步迭代排列。Keccak[r, c, d]函数中使用了两个函数 $\text{pad}(M,n)$ 和 $\text{enc}(x,n)$，这两个函数的功能如下：

$\text{pad}(M,n)$ 表示在 M 后添加一比特“1”，紧接着添加最少的“0”使得添加后的长度是 n 的整数倍。$\text{enc}(x,n)$ 表示整数 x 的 n 比特编码，从最低有效比特到最高有效比特，当 $x = \sum_{i=0}^{n-1} x_i 2^i$ 时，它返回字符串 $x_0, x_1, \cdots, x_{n-1}$。

SHA-3 Keccak 算法根据产生摘要的长度分为四种类型，如下所示。

（1）SHA3-224：$\lfloor \text{Keccak}[r=1024, c=576, d=28] \rfloor_{224}$

（2）SHA3-256：$\lfloor \text{Keccak}[r=1024, c=576, d=32] \rfloor_{256}$

（3）SHA3-384：$\lfloor \text{Keccak}[r=512, c=1088, d=48] \rfloor_{384}$

（4）SHA3-512：$\lfloor \text{Keccak}[r=512, c=1088, d=64] \rfloor_{512}$

9.2 Keccak 算法相关模块 FPGA 设计

通过对算法结构的分析，用 FPGA 实现该算法主要分四个模块：主函数模块、轮函数模块、轮常数模块与缓存模块。Keccak 算法轮函数共需要 5 个计算步骤，共 24 轮迭代。本实例采用一个周期内完成五步运算，这样共需要 24 个周期完成迭代，但考虑到数据读取共需要 1088 位，需要 34 个时钟周期，迭代函数的计算周期也将大于 34。

9.2.1 主函数模块的设计

Keccak 算法状态更新使用的是三维数组上的迭代排列，且所涉及的中间数据指标较多，因此采用选择结构和循环结构进行运算。主函数模块的程序实现，参考程序 9-1。

程序 9-1：

```
p_main : process (clk, rst_n)
begin  -- process p_main
ifrst_n = '0' then                -- 异步 rst_n 信号，低电平有效
            --reg_data<= zero_state;
        for row in 0 to 4 loop
            for col in 0 to 4 loop
                fori in 0 to 63 loop
```

```
                    reg_data(row)(col)(i)<='0';
                end loop;
            end loop;
        end loop;
counter_nr_rounds<= (others => '0');
permutation_computed<='1';
elsifclk'event and clk = '1' then  -- clk上升沿有效
    if (start='1') then
        --reg_data<= zero_state;
        for row in 0 to 4 loop
            for col in 0 to 4 loop
                fori in 0 to 63 loop
                    reg_data(row)(col)(i)<='0';
                end loop;
            end loop;
        end loop;
        counter_nr_rounds<= (others => '0');
        permutation_computed<='1';
    else
        if(din_buffer_full ='1' and permutation_computed='1') then
            counter_nr_rounds(4 downto 0)<= (others => '0');
            counter_nr_rounds(0)<='1';
            permutation_computed<='0';
            reg_data<= round_out;
        else
            if(counter_nr_rounds< 24 and permutation_computed='0') then
                counter_nr_rounds<= counter_nr_rounds + 1;
                reg_data<= round_out;
            end if;
            if(counter_nr_rounds = 23) then
                permutation_computed<='1';
                counter_nr_rounds<= (others => '0');
            end if;
        end if;
    end if;
end if;
end process p_main;
-- 输入mapping
--容量部分
    i01: for col in 1 to 4 generate
        i02: for i in 0 to 63 generate
            round_in(3)(col)(i)<= reg_data(3)(col)(i);
        end generate;
    end generate;
```

```
        i03: for col in 0 to 4 generate
            i04: for i in 0 to 63 generate
                round_in(4)(col)(i)<= reg_data(4)(col)(i);
            end generate;
        end generate;
    -- 比特部分
    i10: for row in 0 to 2 generate
        i11: for col in 0 to 4 generate
            i12: for i in 0 to 63 generate
                round_in(row)(col)(i)<= reg_data(row)(col)(i) xor
                        (din_buffer_out((row*64*5)+(col*64)+i) and
                        (din_buffer_full and permutation_computed));
            end generate;
        end generate;
    end generate;
    i13: for i in 0 to 63 generate
                round_in(3)(0)(i)<= reg_data(3)(0)(i) xor
                        (din_buffer_out((3*64*5)+(0*64)+i) and
                        (din_buffer_full and permutation_computed));
    end generate;
```

当 reg_data 小于 zero_state 时，执行四个 for 语句，如果时钟为 1，并且 start 为 1 时，reg_data 状态为 zero_state，否则为小于 0 的状态。reg_data>=0，则 permutation_computed<='1'；如果 din_buffer_full ='1' and permutation_computed='1'，则 counter_nr_rounds<= counter_nr_rounds + 1。

9.2.2 轮函数模块设计

轮函数运算为 SHA-3 算法的核心模块，负责完成算法立体功能，对于不同的 Keccak 算法，仅需改变此模块的设计。轮函数的模块结构如图 9.4 所示。

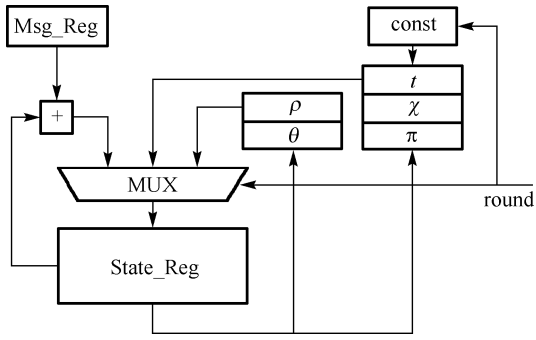

图 9.4 Keccak 算法轮函数结构

轮函数模块的程序实现，参考程序 9-2。

程序 9-2：

```
    entitykeccak_round is
    port (
    round_in     : in  k_state;
```

```
round_constant_signal    : in std_logic_vector(63 downto 0);
round_out    : out k_state);
endkeccak_round;
architecture Behavioral of keccak_round is
signal theta_in,theta_out,pi_in,pi_out,rho_in,rho_out,chi_in,chi_out,
            iota_in,iota_out : k_state;
signalsum_sheet: k_plane;
begin  --Rtl
- 顺序 theta,, pi， rho， chi， iota
theta_in<=round_in;
pi_in<=rho_out;
rho_in<=theta_out;
chi_in<=pi_out;
iota_in<=chi_out;
round_out<=iota_out;
- chi
i0000: for y in 0 to 4 generate
    i0001: for x in 0 to 2 generate
        i0002: for i in 0 to 63 generate
            chi_out(y)(x)(i)<=chi_in(y)(x)(i) xor  ( not(chi_in (y)(x+1)(i))
                            and chi_in (y)(x+2)(i));
        end generate;
    end generate;
end generate;
    i0011: for y in 0 to 4 generate
        i0021: for i in 0 to 63 generate
            chi_out(y)(3)(i)<=chi_in(y)(3)(i) xor  ( not(chi_in (y)(4)(i))
                            and chi_in (y)(0)(i));
                end generate;
    end generate;
        i0012: for y in 0 to 4 generate
        i0022: for i in 0 to 63 generate
            chi_out(y)(4)(i)<=chi_in(y)(4)(i) xor  ( not(chi_in (y)(0)(i))
                            and chi_in (y)(1)(i));
                end generate;
    end generate;
```

从轮函数模块设计代码中可以看出，输入为 k_state，轮控制信号为 std_logic_vector，其中执行的顺序为 theta, pi, rho, chi, iota。

9.2.3　轮常数模块的设计

在不同的模块中，会有不同的轮常数。根据具体情况详细罗列了所有情况，使用 when 语句描述，参考程序如程序 9-3 所示。

程序 9-3：

```
begin  --Rtl
round_constants : process (round_number)
begin
    caseround_number is
   when "00000" =>round_constant_signal<= X"0000000000000001" ;
    when "00001" =>round_constant_signal<= X"0000000000008082" ;
    when "00010" =>round_constant_signal<= X"800000000000808A" ;
    when "00011" =>round_constant_signal<= X"8000000080008000" ;
    when "00100" =>round_constant_signal<= X"000000000000808B" ;
    when "00101" =>round_constant_signal<= X"0000000080000001" ;
    when "00110" =>round_constant_signal<= X"8000000080008081" ;
    when "00111" =>round_constant_signal<= X"8000000000008009" ;
    when "01000" =>round_constant_signal<= X"000000000000008A" ;
    when "01001" =>round_constant_signal<= X"0000000000000088" ;
    when "01010" =>round_constant_signal<= X"0000000080008009" ;
    when "01011" =>round_constant_signal<= X"000000008000000A" ;
    when "01100" =>round_constant_signal<= X"000000008000808B" ;
    when "01101" =>round_constant_signal<= X"800000000000008B" ;
    when "01110" =>round_constant_signal<= X"8000000000008089" ;
    when "01111" =>round_constant_signal<= X"8000000000008003" ;
    when "10000" =>round_constant_signal<= X"8000000000008002" ;
    when "10001" =>round_constant_signal<= X"8000000000000080" ;
    when "10010" =>round_constant_signal<= X"000000000000800A" ;
    when "10011" =>round_constant_signal<= X"800000008000000A" ;
    when "10100" =>round_constant_signal<= X"8000000080008081" ;
    when "10101" =>round_constant_signal<= X"8000000000008080" ;
    when "10110" =>round_constant_signal<= X"0000000080000001" ;
    when "10111" =>round_constant_signal<= X"8000000080008008" ;
    when others =>round_constant_signal<=(others => '0');
end case;
end process round_constants;
round_constant_signal_out<=round_constant_signal;
end Behavioral;
```

根据以上代码分析可知：当轮常数为 00000 时，输出为 0000000000000001；当轮常数为 00001，输出为 0000000000008082；当轮常数为 00010，输出为 800000000000808A；当轮常数为 00011，输出为 8000000080008000；其他情况类似，不再赘述。

9.2.4 缓存模块设计

FPGA 中的缓存模块设计由数据预处理模块及数据缓存模块组成，数据预处理模块完成将 16 位的数据整合为数据输出以及对数据实施 FFT 变换等，数据缓存模块主要由 FIFO（先进先出）缓存器组成。缓存模块的程序实现，参考程序 9-4。

程序 9-4：

```
begin  -- Rtl
-- buffer
p_main : process (clk, rst_n)
variablecount_out_words:integer range 0 to 4;
begin  -- process p_main
ifrst_n = '0' then               -- 异步 rst_n 信号，低电平有效
buffer_data<= (others => '0');
count_in_words<= (others => '0');
count_out_words :=0;
buffer_full<='0';
mode<='0';
dout_buffer_out_valid<='0';
elsifclk'event and clk = '1' then  -- rising clk edge
        if(last_block ='1' and ready='1') then
        mode<='1';
    end if;
    -- 输入模块
    if (mode='0') then
        if(buffer_full='1' and ready ='1')  then
            buffer_full<='0';
            count_in_words<= (others=>'0');
                    else
                        if (din_buffer_in_valid='1' and buffer_full='0') then
                            -- shift buffer
                        fori in 0 to 14 loop
                    buffer_data( 63+(i*64) downto 0+(i*64) )<=buffer_data
                            ( 127+(i*64) downto 64+(i*64) );
                                end loop;
                        -- 插入新的输入
                        buffer_data(1023 downto 960) <= din_buffer_in;
                        if (count_in_words=15) then
                            -- buffer full ready for being absorbed by the
                                    permutation
                            buffer_full<= '1';
                            count_in_words<= (others=>'0');

                        else
                            count_in_words<= count_in_words + 1;
                        end if;
            --  end if;
            end if;
        end if;
```

```
        else
            -- 输出模块
            dout_buffer_out_valid<='1';
            if(count_out_words=0) then
                buffer_data(255 downto 0) <= dout_buffer_in;
                count_out_words:=count_out_words+1;
                dout_buffer_out_valid<='1';
                -- i 从 0 到 2 的循环
                --缓冲数据 63+(i*64) downto 0+(i*64) )<=buffer_data(127+(i*64)
                        downto 64+(i*64) );
                -- end loop; 循环结束
            else
                if(count_out_words<4) then
                    count_out_words:=count_out_words+1;
                    dout_buffer_out_valid<='1';
                    fori in 0 to 2 loop
                buffer_data( 63+(i*64) downto 0+(i*64) )<=buffer_data
                            ( 127+(i*64) downto 64+(i*64) );
                    end loop;
                else
                    dout_buffer_out_valid<='0';
                    count_out_words:=0;
                    mode<='0';
                end if;
            end if;
        end if;
    end if;
    end process p_main;
```

以上缓存模块以 begin 开始的代码为主要的程序，rst_n 是异步信号，clk 是上升沿有效，if(last_block ='1' and ready='1')…end 为输入模块。

9.3 Keccak 算法工程实现

1．创建 ISE 工程

打开 ISE 开发环境，选择菜单 File→New Project，建立一个新工程，然后设置顶层文件模块名、存储目录和设计方式。接着选择器件，这里选择 Xilinx 公司的 Virtex4 来实现。最后再选择第三方仿真软件 ModelSim 和 Verilog 语言进行仿真测试。新工程的建立如图 9.5～图 9.7 所示。

2．编写设计代码

在 9.2 节中详细叙述了 Keccak 算法的关键模块设计，上述模块在 ISE 中的具体实现如图 9.8 所示。

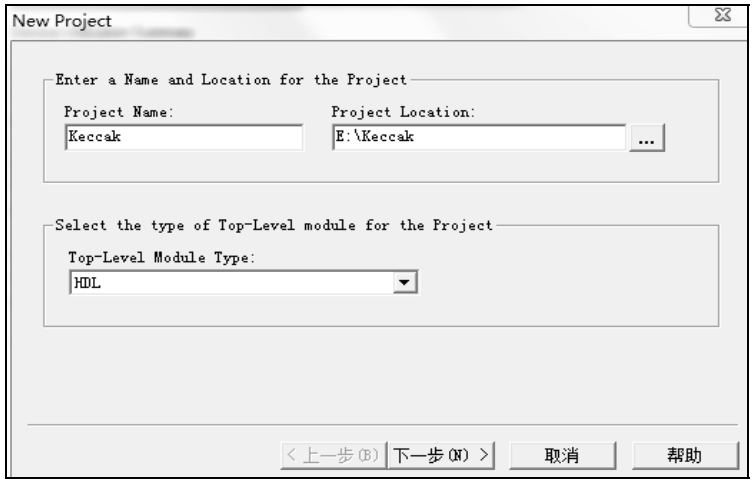

图 9.5　Keccak 算法工程文件的建立

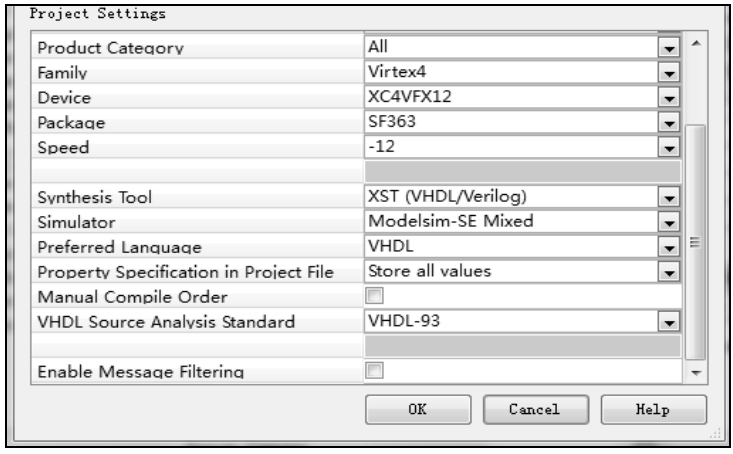

图 9.6　器件的选择和工具的使用

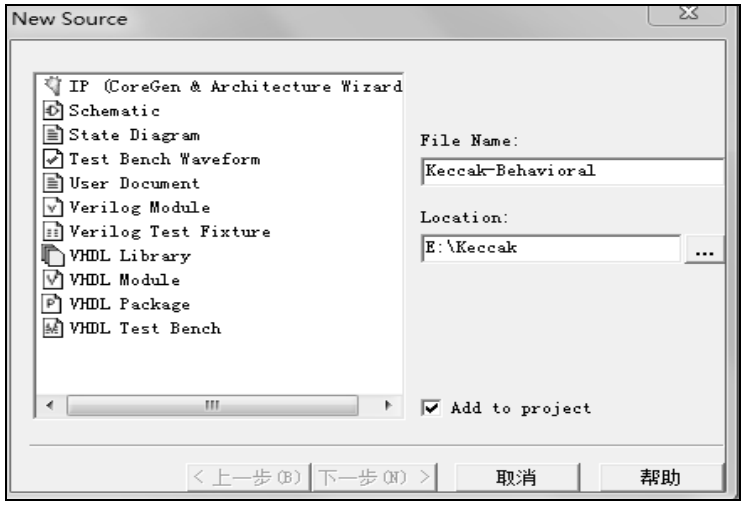

图 9.7　新建工程

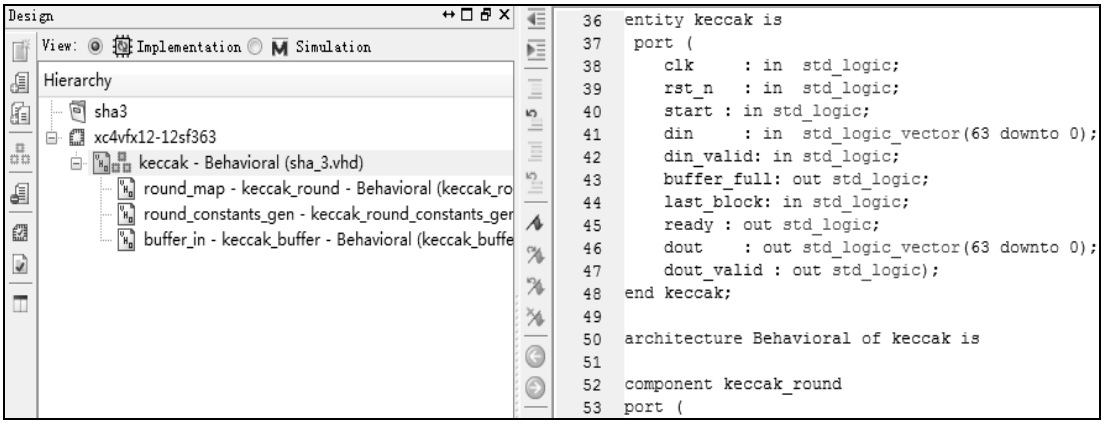

图 9.8 打开工程

3．工程验证

（1）根据以上编写的程序，编写该程序的测试文件，如图 9.9 所示。

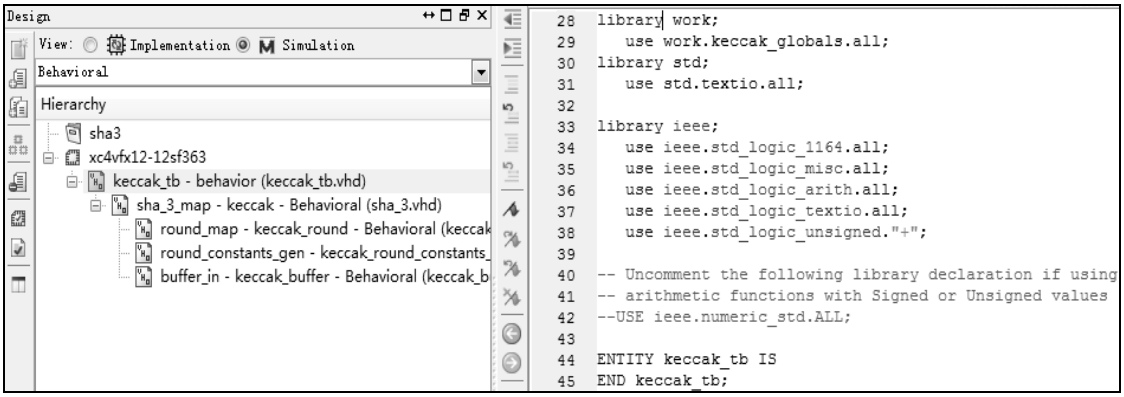

图 9.9 Keccak 算法测试文件

（2）利用 ModelSim 仿真软件，对该算法进行仿真测试。选中主程序 keccak_tb-behavio，在 ModelSim 中选中 Simulate Behavioral Model 进行仿真，如图 9.10 和图 9.11 所示。

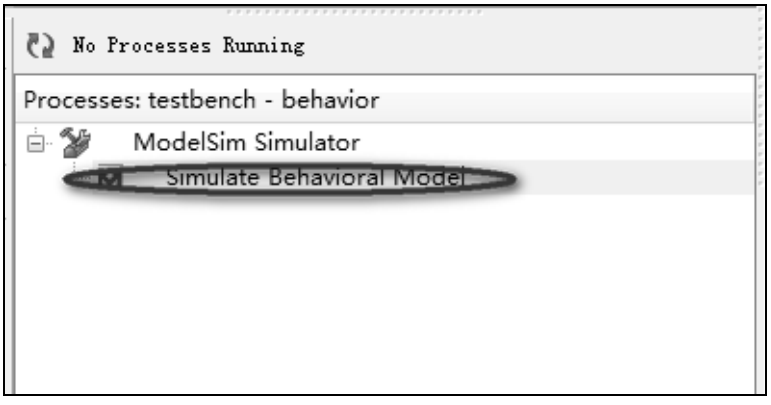

图 9.10 调用 ModelSim

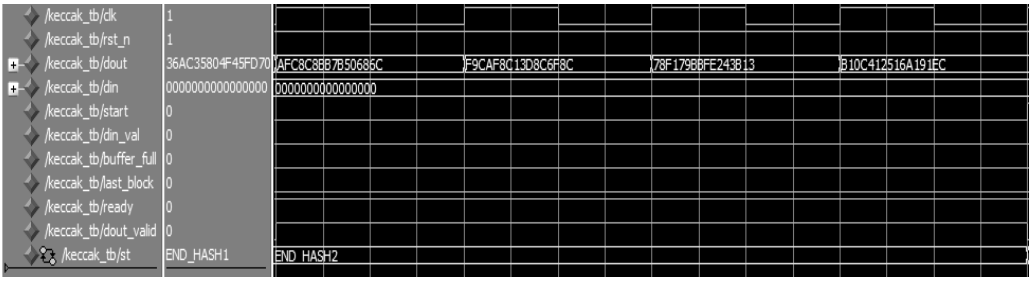

图 9.11　Keccak 算法仿真测试结果

9.4 效果测试

9.3 节中的仿真与测试结果，可以验证整个设计的合理性与正确性。还要在芯片上对整个设计进行综合测试分析。

本实例仍然使用 ModelSim 对算法进行仿真，以验证算法运行是否正确。在 Keccak 算法中，输入为 2048 位 16 进制的明文如下：

```
CD0E32EA2DD1F44C 4F06382FF1729E49 6896717839F3F90C B5B76171F9D7AFDD
85239E6795293C68 5ED855CB69F4723B 7699C3B342CAF64B 3A6DBC59E23EC270
CB1E289A806AA1E4 7BBB8178466AB6C8 54175798D2DFF59F 9247CCC5B56DA8D1
2428186BF73DEB2A 2FD7051D917B3058 8B37740FF1D9BC70 446ED79C7A0B296A
018000E10149E2A0 0000000000000000 0000000000000000 0000000000000000
0000000000000000 0000000000000000 0000000000000000 0000000000000000
0000000000000000 0000000000000000 0000000000000000 0000000000000000
0000000000000000 0000000000000000 0000000000000000 0000000000000000
```

其结果为 256 位 16 进制的 Hash 运算输出：

```
5A544AC3234A0658 3B08E95B86568B9F CC4FA411AA937F2D 9F92502685125036
```

输入输出的仿真波形如图 9.12 所示。

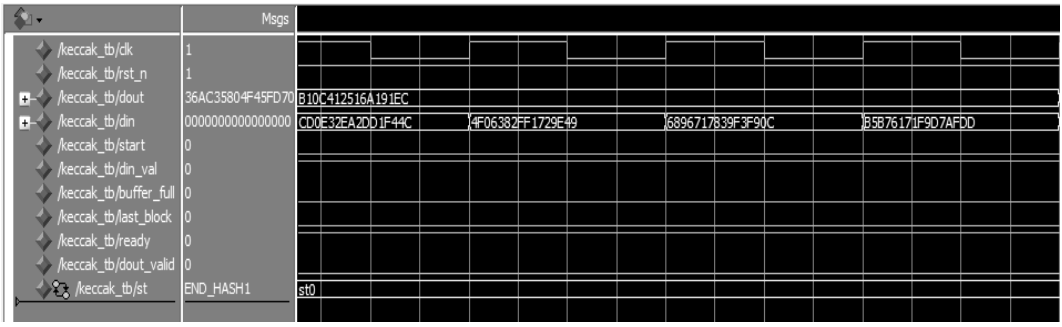

图 9.12　输入 Keccak 算法测试的明文

下面对该工程的时序、运行效率、资源占用情况进行分析。在 Xilinx 公司的 4vfx12 FPGA 元器件上，使用 ISE 的 XST 分析工具进行综合，综合结果如图 9.13 和图 9.14 所示。

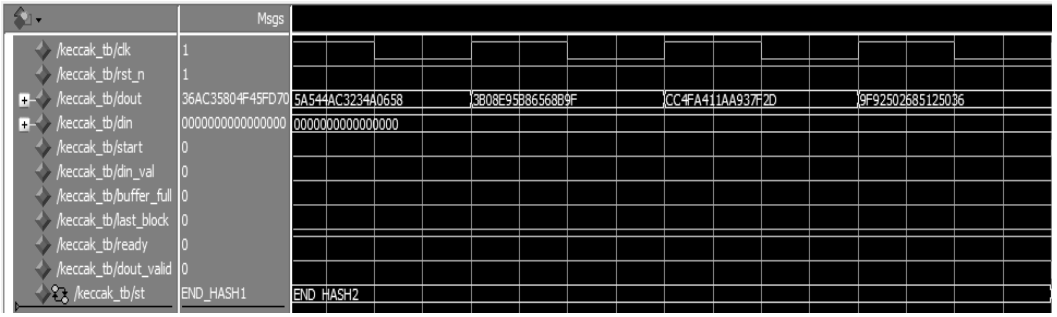

图 9.13　Keccak 算法输出的 Hash 值

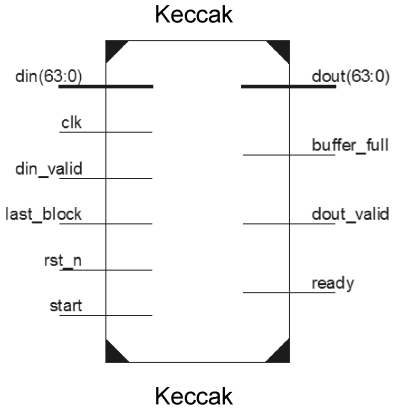

图 9.14　Keccak 算法综合后的外部器件

资源占用情况分析报告如表 9.1 所示。

表 9.1　Keccak 算法资源占用情况

Selected Device	4vfx12sf363-12
Number of Slices	2910 out of 5472 53%
Number of Slice Flip Flops	2745 out of 10944 25%
Number of 4 input LUTs	5590 out of 10944 51%
Number of IOs	136
Number of bonded IOBs	136 out of 240 56%
IOB Flip Flops	4
Number of GCLKs	1 out of 32 3%

时序分析报告如下。

Timing Summary:

Speed Grade: -12

Minimum period: 3.674 ns (Maximum Frequency: 272.177 MHz)

Minimum input arrival time before clock: 4.620 ns

Maximum output required time after clock: 4.528 ns

Maximum combinational path delay: No path found

由时序分析报告可知：Keccak 算法的最高工作频率为 272.177 MHz。分析程序可知，Keccak 算法的数据吞吐率为 12 Gbps。

由以上分析报告可以看出，整个工程在 Xilinx 公司的 4vfx12 芯片上使用的查找表数为 941，占芯片资源的 51%；使用了 136 个引脚，占引脚资源的 56%。可以看出使用的外部器件并不多，占用资源也不多，速度很快。因此，运算性能比较乐观。

9.5 本章小结

本章介绍了 Keccak 算法的设计思想、原理，描述了 Keccak 算法 FPGA 实现的基本操作流程，分别说明了 Keccak 算法 FPGA 实现的主要模块。通过仿真更加直观准确地了解到 Keccak 算法的运算性能。Keccak 算法作为 SHA-3 的获胜者，已经引起人们广泛的关注，对它的分析也日益增多，Keccak 基于比特级的置换将会得到更为深入的研究，Sponge 结构的特点和进一步应用也会受到同样关注。

第 10 章　SM3 算法 FPGA 实现

为了满足电子认证服务等应用需求，中国国家密码管理局于 2010 年 12 月发布了 SM3 Hash 算法。该算法适用于商用密码应用中的数字签名和验证、消息认证码的生成与验证以及随机数的生成，可满足多种密码应用的安全需求。SM3 算法能够对任何长度小于 2^{64}bit 的数据进行计算，输出长度为 256 bit 的 Hash 值。

10.1　SM3 算法原理

10.1.1　算法描述

对长度为 $l(l<2^{64})$ 比特的消息 m，SM3 杂凑算法经过填充和迭代压缩，生成杂凑值，杂凑值长度为 256 比特。

（1）填充

假设消息 m 的长度为 l 比特。首先将比特“1”添加到消息的末尾，再添加 k 个“0”，k 是满足 $l+1+k=448 \bmod 512$ 的最小的非负整数。然后再添加一个 64 位比特串，该比特串是长度 l 的二进制表示。填充后消息 m'的比特长度为 512 的倍数。

例如：对消息“*abc*”首先将它转换成位字符串为 01100001 01100010 01100011，其长度 l=24，经填充得到 512 位的比特串

$$01100001\ \ 01100010\ \ 011000111\ \overbrace{00\cdots00}^{423\text{bit}}\ \overbrace{\underbrace{00\cdots011000}_{l\text{的二进制表示}}}^{64\text{bit}}$$

（2）迭代压缩

① 迭代过程：

将填充后的消息 m' 按 512 比特进行分组

$$m'=B^{(0)}B^{(1)}\cdots B^{(n-1)}$$

其中，$n=(l+k+65)/512$。

对 m' 按下列方式迭代

for $i=0$ to $n-1$

$$V^{(i+1)}=\mathrm{CF}(V^{(i)},B^{(i)})$$

endfor

其中，CF 是压缩函数，$V^{(0)}$ 为 256 比特初始值 IV，$B^{(i)}$ 为填充后的消息分组，迭代压缩的结果为 $V^{(n)}$。

② 消息扩展：

将消息分组 $B^{(i)}$ 按以下方法扩展生成 132 个字 $W_0,W_1,\cdots,W_{67},W_0',W_1',\cdots,W_{63}'$，用于压缩函数 CF。

第一步，将消息分组 $B^{(i)}$ 划分为 16 个字 $W_0,W_1,\cdots,W_{15}$；

第二步，进行下列循环

for $j=16$ to 67

$W_j \leftarrow P_1(W_{j-16} \oplus W_{j-9} \oplus (W_{j-3} <<< 15)) \oplus (W_{j-13} <<< 7) \oplus W_{j-6}$

endfor

第三步，进行下列循环

for $j=16$ to 63

$W_j^{'} = W_j \oplus W_{j+4}$

endfor

（3）压缩函数

令 A,B,C,D,E,F,G,H 为字寄存器，SS1,SS2,TT1,TT2 为中间变量,压缩函数 $V^{(i+1)}=\mathrm{CF}(V^{(i)},B^{(i)}),0\leqslant i\leqslant n-1$。计算过程描述如下。

$\mathrm{ABCDEFGH} \leftarrow V^{(j)}$

for $j=0$ to 63

$\mathrm{SS1} \leftarrow (\mathrm{A} <<< 12) + \mathrm{E} + (T_j <<< j) <<< 7$

$\mathrm{SS2} \leftarrow \mathrm{SS1} \oplus (\mathrm{A} <<< 12)$

$\mathrm{TT1} \leftarrow \mathrm{FF}_j(\mathrm{A},\mathrm{B},\mathrm{C}) + \mathrm{D} + \mathrm{SS2} + W_j^{'}$

$\mathrm{TT2} \leftarrow \mathrm{GG}_j(\mathrm{E},\mathrm{F},\mathrm{G}) + \mathrm{H} + \mathrm{SS1} + W_j$

$\mathrm{D} \leftarrow \mathrm{C}$

$\mathrm{C} \leftarrow \mathrm{B} <<< 9$

$\mathrm{B} \leftarrow \mathrm{A}$

$\mathrm{A} \leftarrow \mathrm{TT1}$

$\mathrm{H} \leftarrow \mathrm{G}$

$\mathrm{G} \leftarrow \mathrm{F} <<< 19$

$\mathrm{F} \leftarrow \mathrm{E}$

$\mathrm{E} \leftarrow P_0(\mathrm{TT2})$

endfor

$V^{(i+1)} \leftarrow \mathrm{ABCDEFGH} \oplus V^{(i)}$

其中，字的存储为大端（big-endian）格式。

（4）杂凑值

$\mathrm{ABCDEFGH} \leftarrow V^{(i+1)}$

输出 256 比特的杂凑值 $y=\mathrm{ABCDEFGH}$。

10.1.2 常数与函数

（1）初始值

IV = 7380166f 4914b2b9 172442d7 da8a0600 a96f30bc 163138aa e38dee4d b0fb0e4e

（2）常量

$$T_j = \begin{cases} 79cc4519 & 0 \leqslant j \leqslant 15 \\ 7a879d8a & 16 \leqslant j \leqslant 63 \end{cases}$$

（3）布尔函数

$$FF_j(X,Y,Z)\begin{cases} X \oplus Y \oplus Z & 0 \leqslant j \leqslant 15 \\ (X \wedge Y) \vee (X \wedge Z) \vee (Y \wedge Z) & 16 \leqslant j \leqslant 63 \end{cases}$$

$$GG_j(X,Y,Z)\begin{cases} X \oplus Y \oplus Z & 0 \leqslant j \leqslant 15 \\ (X \wedge Y) \vee (\neg X \wedge Z) & 16 \leqslant j \leqslant 63 \end{cases}$$

式中，X、Y、Z 为字，$\wedge$、$\vee$、$\neg$ 和 $\oplus$ 分别表示与操作、或操作、非操作和异或操作。

（4）置换函数

$$P_0(X) = X \oplus (X <<< 9) \oplus (X <<< 17)$$

$$P_1(X) = X \oplus (X <<< 15) \oplus (X <<< 23)$$

式中，X 为字。

10.2 SM3 算法相关模块 FPGA 设计

通过对算法结构的分析发现，用 FPGA 实现该算法主要需要四个模块：控制单元模块、消息扩展模块、迭代压缩模块和结果输出模块。SM3 算法各个模块示意图如图 10.1 所示。

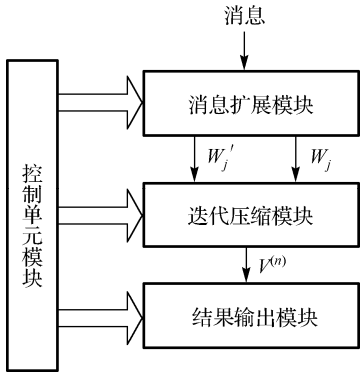

图 10.1 SM3 算法的关键模块

10.2.1 控制单元设计

SM3 算法的控制单元比较简单，设计了简单的状态机来完成整个 Hash 运算。状态机总共包含了 70 个状态，这 70 个状态控制了 SM3 算法的三大运算模块（消息扩展模块、迭代压缩

模块和结果输出模块)并完成 Hash 运算。设计中通过设置复位键 rst_i 和不同的状态变量 cmd_i 来实现对不同模块的控制。控制单元的程序实现，参考程序 10-1。

程序 10-1：

```
always @ (posedge clk_i)
    begin
        if (rst_i)
            cmd <= 'b0;
        else
          if (cmd_w_i)
            cmd[2:0] <= cmd_i[2:0];     // busy bit , can't write
          else
            begin
            cmd[3] <= busy;         // update busy bit
            if (~busy)
                cmd[1:0] <= 2'b00;  // hardware auto clean R/W bits
        end
    end
...
if (rst_i)
begin
round <= 'd0;
busy <= 'b0;
 W0 <= 'b0;                 W1 <= 'b0;                W2 <= 'b0;
 W3 <= 'b0;                 W4 <= 'b0;                W5 <= 'b0;
 W6 <= 'b0;                 W7 <= 'b0;                W8 <= 'b0;
 W9 <= 'b0;                W10 <= 'b0;               W11 <= 'b0;
W12 <= 'b0;                W13 <= 'b0;               W14 <= 'b0;
W15 <= 'b0;                W16 <= 'b0;               W17 <= 'b0;
W18 <= 'b0;                W19 <= 'b0;                Wt <= 'b0;
  A <= 'b0;                  B <= 'b0;                 C <= 'b0;
  D <= 'b0;                  E <= 'b0;                 F <= 'b0;
  G <= 'b0;                  H <= 'b0;                H0 <= 'b0;
 H1 <= 'b0;                 H2 <= 'b0;                H3 <= 'b0;
 H4 <= 'b0;                 H5 <= 'b0;                H6 <= 'b0;
 H7 <= 'b0;
end
else
  ...
if (cmd[1])
    begin
    W0 <= text_i;       busy <= 'b1;    round <= round_plus_1;
   case (cmd[2])
        1'b0: ···
       1'b1: ···
```

```
    endcase
else
    begin   // IDLE
        round <= 'd0;
            end
```

在程序 10-1 中，在时钟上升沿时刻，rst_i 复位键高位有效，2 个时钟周期以后，设置成低位，然后通过对 cmd 值进行运算，达到对不同模块进行控制的目的。cmd[3:0]依次从高位到低位控制 busy 操作、循环轮数操作、写命令操作和读命令操作。当 cmd[3]=0 时，进行 idle 操作，不加载 busy；当 cmd[3]=1 时，进行 busy 操作。当 cmd[2]=0 时，进行第 1 轮操作，可以得到一个 256 比特的输出值；当 cmd[2]=1 时，进行如下的循环次数操作：将上一轮的输出值作为该次循环的输入初始值，依次循环，最终完成对整个明文的 SM3 运算操作。当 cmd[1]=0 时，不进行写操作；cmd[1]=1 时，将数据写入寄存器，并进行消息的扩展。当 cmd[0]=0 时，不进行读操作；cmd[0]=1 时，将最终结果进行读操作指令，并进行显示。

10.2.2 消息扩展模块设计

根据控制单元给出的状态消息扩展模块，按照算法要求的步骤完成消息的扩展。具体过程如下：在控制状态机的前四个状态，消息扩展分别完成初始化W_0,W_1,W_2,W_3，本实例设计的数据端口为 32bit 宽，这样设计的初衷是便于将设计的模块方便地挂载到目前常用的数据总线上，数据是按照每个时钟周期 32bit 进入 Hash 单元。从第 6 个状态到第 16 个状态，分别用进入的消息数据值完成$W_5 \rightarrow W_{15}$的初始化过程，同时完成$W_0' \rightarrow W_{10}'$的初始化过程。从第 17 个状态到第 22 个状态，开始计算$W_{16} \rightarrow W_{19}$的数据，并完成$W_{11}' \rightarrow W_{15}'$的初始化过程，至此，前 16 轮操作完成。从第 23 个状态到第 68 个状态，完成$W_{20} \rightarrow W_{67}$的初始化，同时完成$W_{16}' \rightarrow W_{63}'$的操作。

在硬件设计中，用了 16 个 32bit 的寄存器来存储 16 个字W，因为每一轮里面，用到的W都可以用 16 个W来生成，16 个W刚好构成一个移位寄存器结构，每次用新产生的 W 更新W_{15}，通过这样的移位寄存器结构，就可以产生其余的W和W'。

在 Verilog HDL 中使用 case 语句实现该消息模块的输入，第 16 个状态循环完成该模块。第 22 个状态完成前 16 步的消息迭代，从第 23 个状态到第 69 个状态，完成后面 48 步的消息迭代过程。消息扩展模块的程序实现，参考程序 10-2。

程序 10-2：

```
case (round)
    'd0:
    begin
      if (cmd[1])  begin
        W0 <= text_i;     busy <= 'b1;     round <= round_plus_1;
            ...
            end
    'd1:
      begin
            W1 <= text_i;
```

```
      round <= round_plus_1;
    end
'd2:
  begin
       W2 <= text_i;
     round <= round_plus_1;
        end
'd3:
    begin
        W3 <= text_i;
      round <= round_plus_1;
         end
'd4:
   begin
         W4 <= text_i;
        round <= round_plus_1;
         end
'd5:
   begin
         W5 <= text_i;
        round <= round_plus_1;
         Wj1 <= W0;
         Wj2 <= W0 ^ W4;
          Tj <= 'SM3_T0;
          end
'd6:
    begin
          W6 <= text_i;
          round <= round_plus_1;

          Wj1 <= W1;    Wj2 <= W1 ^ W5;      Tj <= next_Tj;
            A <= TT1;      B <= A;             C <= next_C;
            D <= C;        E <= next_E;        F <= E;
            G <= next_G;  H <= G;
          end
'd7:
    begin
        ...
        end
'd8:
   begin
        W8 <= text_i;
         ...
        end
```

```
        'd9:
           begin
                W9 <= text_i;
                 ...
                end
        'd10:
           begin
                W10 <= text_i;
                 ...
                end
        'd11:
           begin
                W11 <= text_i;
                 ...
                 end
        'd12:
           begin
                W12 <= text_i;
                 ...
                end
        'd13:
            begin
                W13 <= text_i;
                 ...
                end
        'd14:
            begin
                W14 <= text_i;
                 ...
               end
        'd15:
            begin
                 W15 <= text_i;
                  ...
             end
    'd16:
          begin
            Wj1 <= W11;    Wj2 <= W11 ^ W15;
            Wt <= W1 ^ W8 ^ {W14[16:0],W14[31:17]};
            W16 <= {Wt ^ {Wt[16:0],Wt[31:17]} ^ {Wt[8:0],Wt[31:9]}} ^
                    {W3[24:0],W3[31:25]} ^ W10;
            Tj <= next_Tj;     A <= TT1;
            B <= A;            C <= next_C;
            D <= C;            E <= next_E;
```

```
        F <= E;           G <= next_G;
        H <= G;           round <= round_plus_1;
        end
    'd17:
      begin
       Wj1 <= W12;   Wj2 <= W12 ^ W16;
       Wt <= W2 ^ W9 ^ {W15[16:0],W15[31:17]};
       W17 <= {Wt ^ {Wt[16:0],Wt[31:17]} ^ {Wt[8:0],Wt[31:9]}} ^
           {W4[24:0],W4[31:25]} ^ W11;
       Tj <= next_Tj;  A <= TT1;
       B <= A;         C <= next_C;
       D <= C;         E <= next_E;
       F <= E;         G <= next_G;
       H <= G;         round <= round_plus_1;
       end
         end
    'd18:
      begin
        ...
       end
        ...
    'd21:
      begin
       Wj1 <= Wj1_1;    Wj2 <= Wj2_1;     Tj <= 32'h9d8a7a87;
       W0 <= W1;        W1 <= W2;         W2 <= W3;
       W3 <= W4;        W4 <= W5;         W5 <= W6;
       W6 <= W7;        W7 <= W8;         W8 <= W9;
       W9 <= W10;       W10 <= W11;       W11 <= W12;
      W12 <= W13;       W13 <= W14;       W14 <= W15;
      W15 <= W16;       W16 <= W17;       W17 <= W18;
      W18 <= W19;       Wt <= W3 ^ W10 ^ {W16[16:0],W16[31:17]};
        A <= TT1;      B <= A;           C <= next_C;
        D <= C;         E <= next_E;      F <= E;
        G <= next_G;    H <= G;          round <= round_plus_1;
end
     'd22, 'd23, 'd24, ···,'d68:
begin
                  W0  <= W1;
                  W1  <= W2;
                  W2  <= W3;
                  W3  <= W4;
                  W4  <= W5;
                  W5  <= W6;
                  W6  <= W7;
```

```
                W7  <= W8;
                W8  <= W9;
                W9  <= W10;
                W10 <= W11;
                W11 <= W12;
                W12 <= W13;
                W13 <= W14;
                W14 <= W15;
                W15 <= W16;
                W16 <= W17;
                W17 <= {Wt ^ {Wt[16:0],Wt[31:17]} ^ {Wt[8:0],Wt[31:9]}} ^
                       {W5[24:0],W5[31:25]} ^ W12;
                Wt <= W3 ^ W10 ^ {W16[16:0],W16[31:17]};
                Wj1 <= Wj1_1;
                Wj2 <= Wj2_1;
                 Tj <= next_Tj;
                  A <= TT1;
                  B <= A;
                  C <= next_C;
                  D <= C;
                  E <= next_E;
                  F <= E;
                  G <= next_G;
                  H <= G;
                  round <= round_plus_1;
            end
```

在程序 10-2 中，利用 case 语句来完成循环状态的操作，并运用非阻塞复制操作，为下面的压缩函数模块准备数据。其中 Wj1 和 Wj2 分别用来存放每个状态的W_i'和W_{i+4}'，W_t 完成 $W_{j-16} \oplus W_{j-9} \oplus (W_{j-3} <<< 15)$ 操作。

10.2.3 迭代压缩模块设计

迭代压缩函数作为系统的关键步骤，是 SM3 算法设计的核心，压缩函数的路径是 SM3 算法硬件实现的重点环节。迭代压缩函数包含了 5 个加法、1 个布尔函数和 1 个置换函数。从此分析可以看出，实现系统吞吐量的大小就由此路径来决定，为了降低关键路径的延时，利用 Carry Save Adder（CSA）结构来压缩关键路径中 5 个加法。对于关键路径中的循环移位，由于在硬件设计中，循环移位的工作就是调整信号线的排列顺序，除了会带来布线延迟外，不会对系统的关键路径带来太大的影响。对于布尔函数和置换函数中的异或、与、或操作，在 FPGA 中的实现都是按照查找表的原理来实现，所以这些操作的时间消耗是一致的。迭代压缩模块的程序实现，参考程序 10-3。

程序 10-3：

```
    assign Wj0_1 = W0 ^ W7 ^ {W13[16:0],W13[31:17]};
    assign Wj1_1 = {Wj0_1 ^ {Wj0_1[16:0],Wj0_1[31:17]} ^{Wj0_1[8:0],
```

```
                Wj0_1[31:9]}} ^ {W3[24:0],W3[31:25]} ^ W10;
assign Wj0_4 = W4 ^ W11 ^ {W17[16:0],W17[31:17]};
assign Wj1_4 = {Wj0_4 ^ {Wj0_4[16:0],Wj0_4[31:17]} ^ {Wj0_4[8:0],
                Wj0_4[31:9]}} ^ {W7[24:0],W7[31:25]} ^ W14;
assign Wj2_1 = Wj1_1 ^ Wj1_4;
assign SS1_0 = {A[19:0],A[31:20]} + E + Tj;
assign SS1 = {SS1_0[24:0],SS1_0[31:25]};
assign SS2 = SS1 ^ {A[19:0],A[31:20]};
assign TT1 = FFj_ABC + D + SS2 + Wj2;
assign TT2 = GGj_EFG + H + SS1 + Wj1;
assign next_C = {B[22:0],B[31:23]};
assign next_E = TT2 ^ {TT2[22:0],TT2[31:23]} ^ {TT2[14:0],TT2[31:15]};
assign next_G = {F[12:0],F[31:13]};
assign next_Tj = {Tj[30:0],Tj[31]};
assign SM3_result = {A,B,C,D,E,F,G,H};
assign round_plus_1 = round + 1;
```

在程序 10-3 中，利用若干个 assign 语句来完成该迭代压缩函数模块，该语句的驱动是通过上一部分循环状态中的非阻塞赋值来得到的。

10.2.4　结果输出模块设计

根据控制单元给出的状态，结果输出模块按照算法要求的步骤完成消息的输出。具体过程如下：通过控制状态机对读状态控制键 read_counter 进行赋值，复位键完成对 read_counter 的初始化，分 8 段完成 Hash 值的输出，本实例设计的数据端口为 32bit 宽，这样设计的初衷是便于将设计的模块挂载到目前常用的数据总线上，数据是按照每个时钟周期 32bit 进入 Hash 单元。结果输出模块的程序实现，参考程序 10-4。

程序 10-4：

```
case (round)
    ...
    'd69:
        begin
            A <= TT1 ^ H0;
            B <= A ^ H1;
            C <= next_C ^ H2;
            D <= C ^ H3;
            E <= next_E ^ H4;
            F <= E ^ H5;
            G <= next_G ^ H6;
            H <= G ^ H7;
            round <= 'd0;
            busy <= 'b0;
        end
    default:
```

```
            begin
                round <= 'd0;
                busy <= 'b0;
            end
        ...
    always @ (posedge clk_i)
    begin
        if (rst_i)
        begin
            text_o <= 'b0;
            read_counter <= 'd9;
        end
        else
        begin
            if (cmd[0])
            begin
                read_counter <= 'd9;    // SM3  256/32=8
            end
            else
            begin
            if (~busy)
            begin
                case (read_counter)
                   'd7: text_o <= SM3_result[8*32-1:7*32];
                   'd6: text_o <= SM3_result[7*32-1:6*32];
                   'd5: text_o <= SM3_result[6*32-1:5*32];
                   'd4: text_o <= SM3_result[5*32-1:4*32];
                   'd3: text_o <= SM3_result[4*32-1:3*32];
                   'd2: text_o <= SM3_result[3*32-1:2*32];
                   'd1: text_o <= SM3_result[2*32-1:1*32];
                   'd0: text_o <= SM3_result[1*32-1:0*32];
                   default:text_o <= 'b0;
                endcase
                if (|read_counter)
                    read_counter <= read_counter - 'd1;
            end
            else
            begin
                text_o <= 'b0;
            end
            end
        end
    end
```

在程序 10-4 中，在第 70 个状态循环，得到了最终输出结果。然后利用一个 always 语句，

通过对 cmd[0]的设置，达到输出 Hash 结果的目的。每次输出 32 比特的数据，分 8 轮输出，通过一个计数指令 read_counter 来完成轮数的操作，输出结果存放在 text_o 中。

10.3 SM3 算法工程实现

1．创建 ISE 工程

打开 ISE 开发环境，选择菜单 File→New Project，建立一个新工程，然后设置顶层文件模块名、存储目录和设计方式。接着选择器件，这里选择 Xilinx 的 Virtex4 来实现。最后再选择第三方仿真软件 ModelSim 和 Verilog 语言进行仿真测试。新工程的建立如图 10.2 和图 10.3 所示。

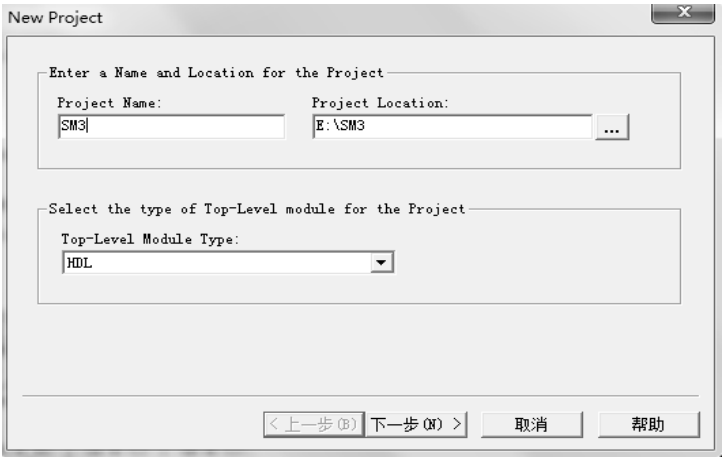

图 10.2　SM3 算法工程文件的建立

图 10.3　器件的选择和工具的使用

2．编写设计代码

在 10.2 节中详细叙述了 SM3 算法的关键模块设计，上面的模块就不做赘述。以下是算法中用到的其他模块代码。

（1）初始值和常量的设计

程序 10-5：

```
'define SM3_H0 32'h7380166f
'define SM3_H1 32'h4914b2b9
'define SM3_H2 32'h172442d7
'define SM3_H3 32'hda8a0600
'define SM3_H4 32'ha96f30bc
'define SM3_H5 32'h163138aa
'define SM3_H6 32'he38dee4d
'define SM3_H7 32'hb0fb0e4e
'define SM3_T0 32'h79cc4519
'define SM3_T1 32'h9d8a7a87
```

在程序 10-5 中，用'define 指令来完成该部分，'define 指令用于定义文本替换的宏命令，类似于 c 语言中的#define 指令。一旦'define 指令被编译通过，则由其规定的宏定义在整个编译过程期间都保持有效。上述的'define 指令定义的初始值和常量宏定义可以在多个文件中使用。

（2）布尔函数和置换函数的设计

程序 10-6：

```
assign FF1_ABC = A ^ B ^ C;
assign FF2_ABC = (A & B) | (A & C) | (B & C);
assign GG1_EFG = E ^ F ^ G;
assign GG2_EFG = (E & F) | (~E & G);
assign FFj_ABC = (round < 'd22) ? FF1_ABC : FF2_ABC;
assign GGj_EFG = (round < 'd22) ? GG1_EFG : GG2_EFG;
```

在 SM3 算法中，布尔函数 $\mathrm{FF}_j(X,Y,Z)$ 是对寄存器 A，B，C 中的数据进行操作，所以设计该布尔函数时，直接调用 A, B, C 中的值，前 16 步是一种运算，后面 48 步是另一种运算，在实现的算法中，从第 6 个状态开始计算 W_j'，到第 22 个状态，前 16 步结束。布尔函数 $\mathrm{GG}_j(X,Y,Z)$ 的实现过程同上所述。

将以上实现的模块代码导入新建的工程中，具体如图 10.4 所示。

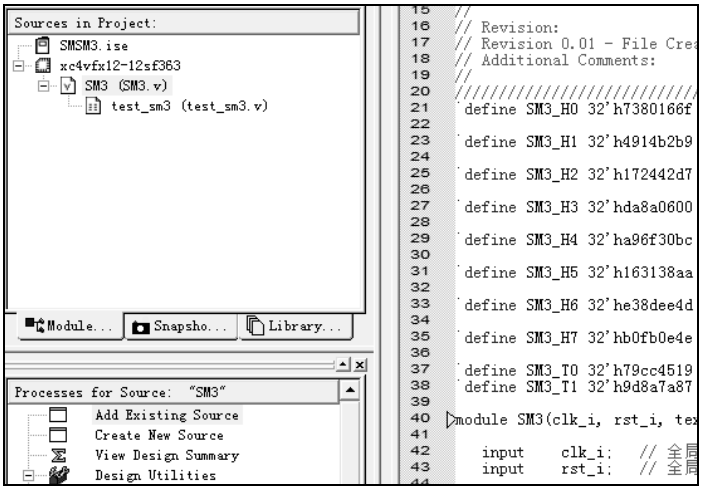

图 10.4 SM3 算法实现

3．工程验证

（1）根据以上编写的程序，编写该程序的测试文件，如图 10.5 和图 10.6 所示。

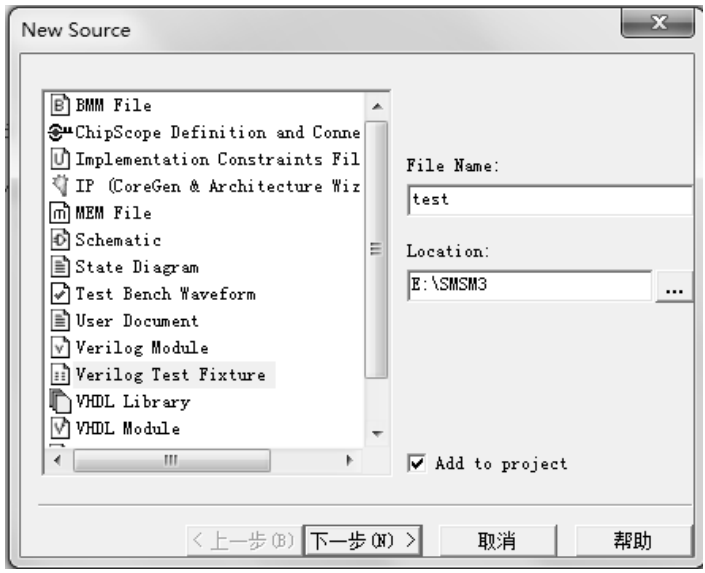

图 10.5　SM3 算法测试文件的建立

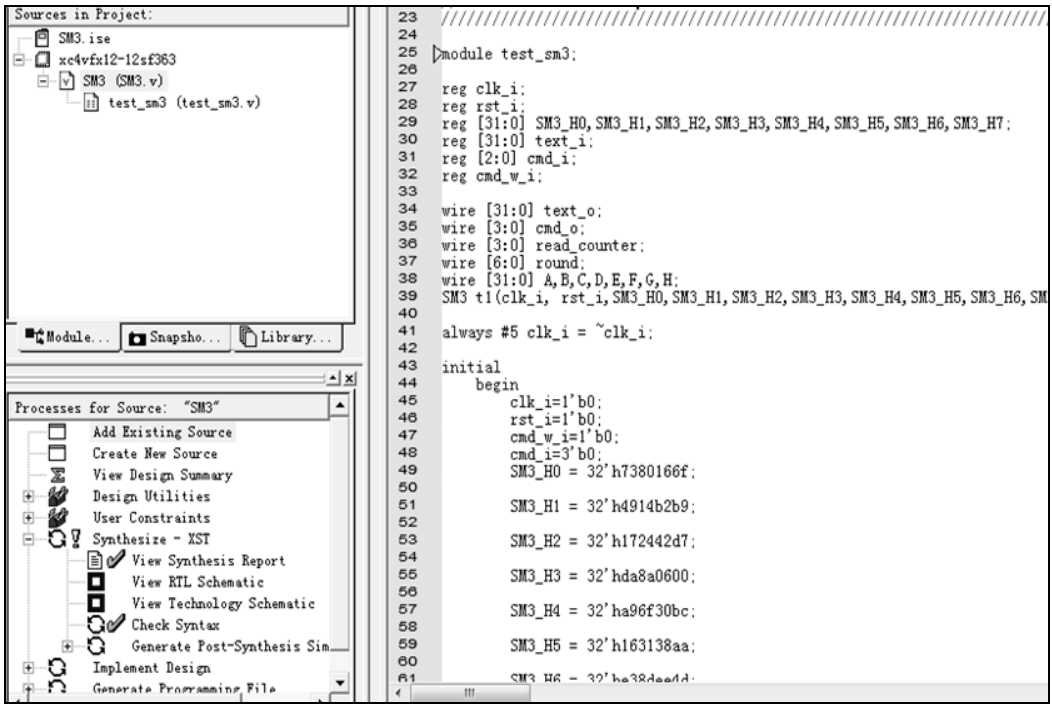

图 10.6　SM3 算法测试文件

（2）根据上一步的测试文件，调用 ModelSim 仿真软件，如图 10.7 所示，对该算法进行仿真测试，仿真结果如图 10.8 所示。

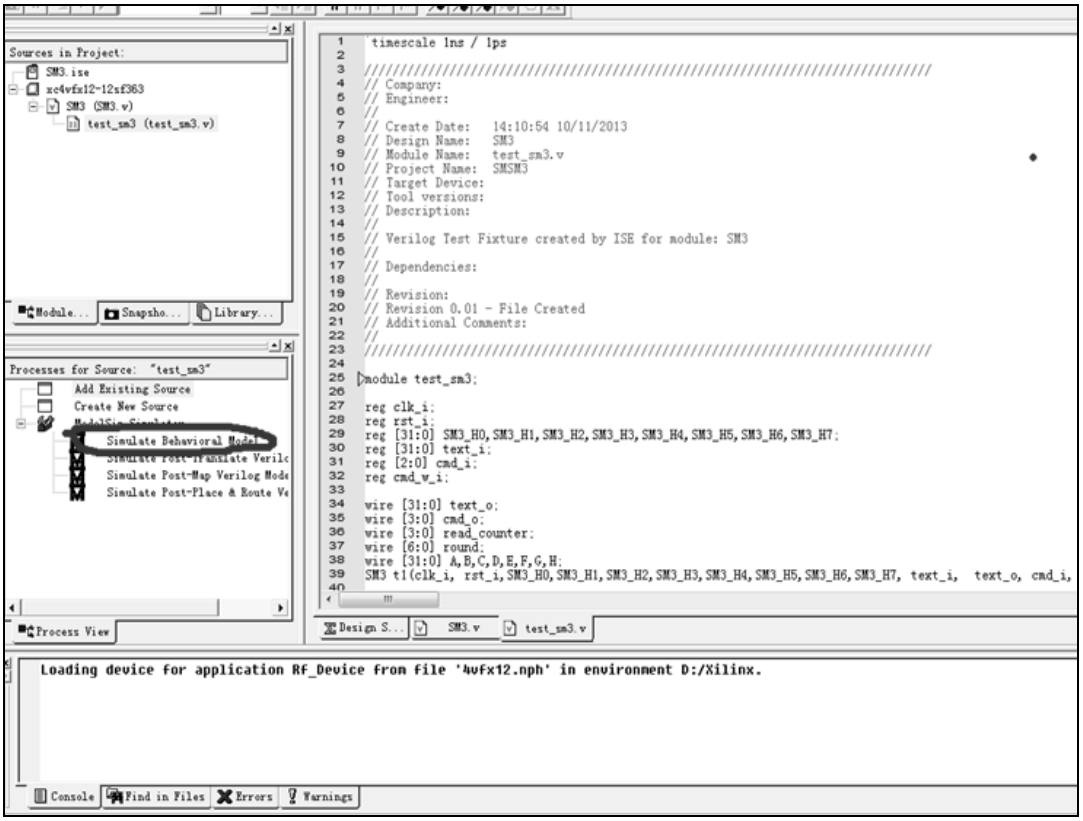

图 10.7 ModelSim 调用示意图

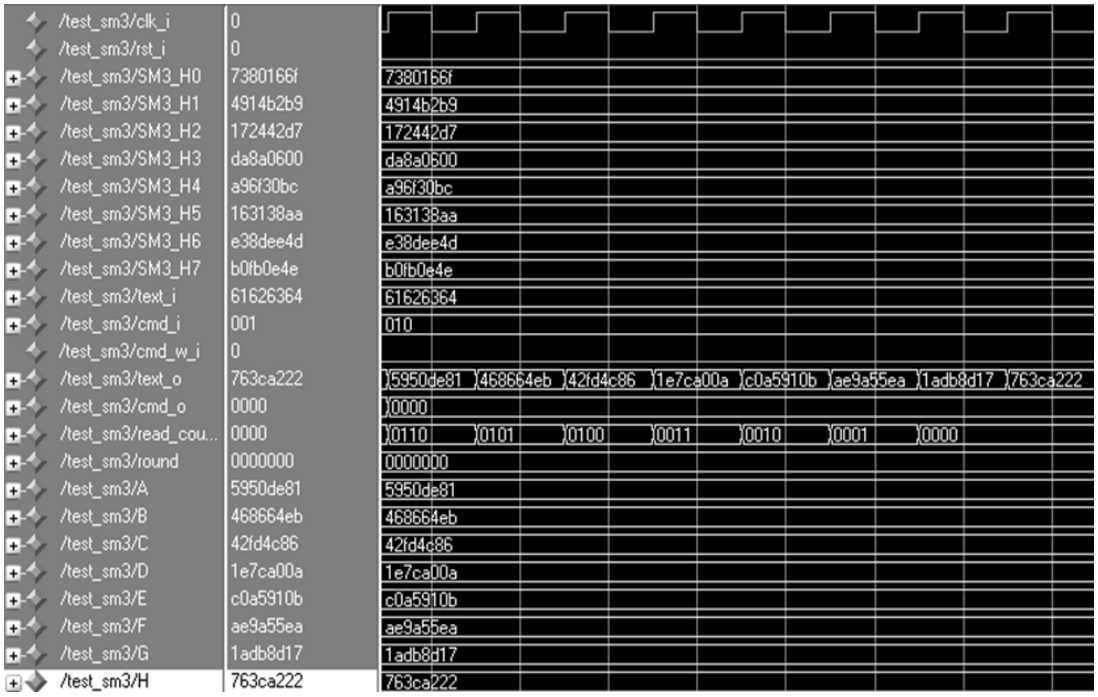

图 10.8 SM3 算法仿真结果

10.4 效果测试

通过功能仿真与系统测试，可以验证整个设计的正确性，为了测试其性能，还需要对整个设计在芯片上的综合结果进行分析。

仿真时，在波形文件中需要给出合适的时钟周期。FPGA 程序设计中，验证算法的正确性和设计合理性的方法之一是对程序进行波形仿真。在本实例中使用 ModelSim 对算法进行仿真，以验证其功能是否正确。例如使用明文（16 进制）为 *abc*，寄存器初始值为 7380166f 4914b2b9 172442d7 da8a0600 a96f30bc 163138aa e38dee4d b0fb0e4e，SM3 算法的 Hash 结果为（16 进制）66c7f0f4 62eeedd9 d1f2d46b dc10e4e2 4167c487 5cf2f7a2 297da02b 8f4ba8e0。SM3 算法过程仿真波形如图 10.9 至图 10.11 所示。

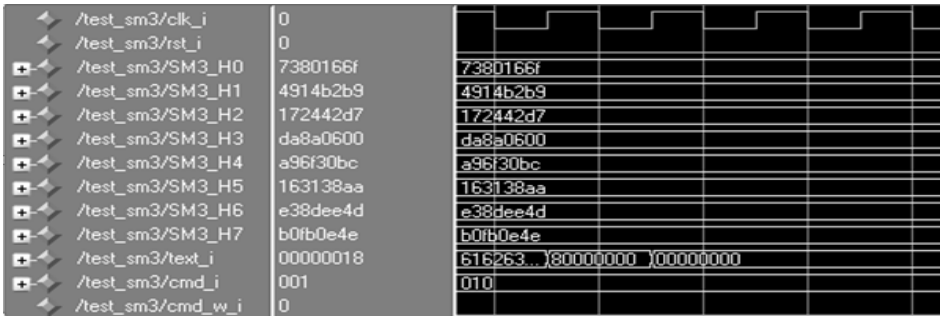

图 10.9　SM3 算法寄存器初始值和输入的明文数据

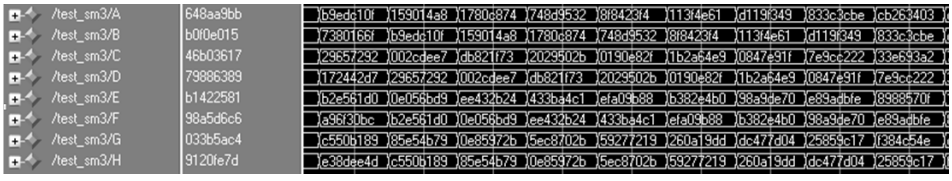

图 10.10　寄存器中间结果

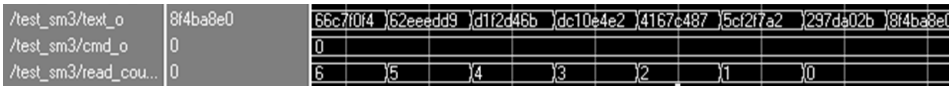

图 10.11　SM3 算法输出的 Hash 结果

下面对该工程的时序、运行效率、资源占用情况进行分析。在 Xilinx 公司的 4vfx12sf363-12 FPGA 元器件上，使用 ISE 的 XST 分析工具进行综合。综合结果如图 10.12 所示。

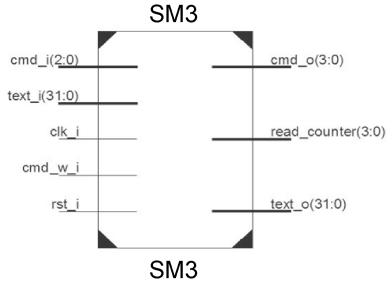

图 10.12　SM3 算法综合后的外部器件

资源占用情况分析报告如表 10.1 所示。

表 10.1 SM3 算法资源占用情况

Selected Device : 4vfx12sf363-12	
Number of Slices	2531 out of 5472 46%
Number of Slice Flip Flops	1329 out of 10944 12%
Number of 4 input LUTs	4723 out of 10944 43%
Number of IOs	78
Number of bonded IOBs	78 out of 240 32%
IOB Flip Flops	1
Number of GCLKs	1 out of 32 3%

时序分析报告如下。

Timing constraint: Default period analysis for Clock 'clk_i'

Clock period: 8.110 ns (frequency: 123.299 MHz)

Total number of paths / destination ports: 19916905 / 2503

由时序分析报告结果可知，SM3 算法的最高工作频率为 123.299 MHz。分析程序可知，每个 512 比特的分组需要经过 70 个状态的处理，共用 70 个时钟。所以，SM3 算法的最高处理速度为 512×123.299/70=901.84 Mbps。

由以上分析报告，可以看出，整个工程在 4vfx12sf363-12 芯片上使用的查找表数为 4723（LUTs），占芯片资源的 43%；使用了 78 个引脚，占引脚资源的 32%。从中可以看出使用的逻辑元器件并不多，达到了性能和效率的折中。

10.5 本章小结

SM3 算法可以把任意长度的输入通过散列算法，变换成固定长度为 256 比特的散列值输出。SM3 算法适用于商用密码应用中的数字签名与验证、消息认证码的生成与验证以及随机数的生成，可满足多种密码应用的安全需求。

本章主要对 SM3 算法进行了介绍，包括基本原理与 FPGA 硬件实现方法，并分别给出了关键模块的实现代码和仿真结果。最后，对该算法进行综合，得出 SM3 算法的工作效率和资源利用情况，使读者对 SM3 算法的 FPGA 实现有了进一步认识。

第 11 章　DSA 数字签名算法 FPGA 实现

11.1 DSA 数字签名原理

数字签名算法（Digital Signature Algorithm，DSA）是 NIST 于 1991 年 8 月 30 日提出，1992 年 5 月 19 日公布，1992 年 12 月 1 日被采纳的一个数字签名标准。该标准的最初版本中建议使用 p 为 512bit 的素数，q 为 p^{-1} 的 160bit 的素因子，后来在众多批评下，NIST 建议使用长为 512～1024 比特的素数 p 。

1．生成参数

（1）基于离散对数困难问题，在 Z_p^* 上选取一个素数 p ，然后选取 q （p–1 的一个大素数因子）。

（2）找出一个阶为 q 的元素 $g \in Z_p^*$ 。

（3）随机生成一个整数 $x(1 < x < q-1)$ ，计算出 $y = g^x \pmod p$ 。

（4）公开参数为 (p,q,g) 。用户的公钥为 y，私钥为 x 。

2．签名算法

秘密选取一随机数 $k \in Z_p^*$ ，

$$\mathrm{Sig}_k(m) = (r,s)$$

其中，$r = (g^k \bmod p) \bmod q, s = (H(m) + xr) \cdot k^{-1} \bmod q$ ，（H 是一个 Hash 函数）；

3．验证算法

签名 (r,s) 真 $\Leftrightarrow (g^{u1} y^{u2} \bmod p) \bmod q = r$

其中，$u1 = H(m)s^{-1} \bmod q, u2 = rs^{-1} \bmod q$ 。

DSA 验证算法正确性的证明如下。

$$\begin{aligned}
&(g^{u1} y^{u2} \bmod p) \bmod q \\
&= (g^{H(m)s^{-1}} y^{rs^{-1}} \bmod p) \bmod q \\
&= (g^{H(m)s^{-1}} g^{xrs^{-1}} \bmod p) \bmod q \\
&= (g^{[H(m)+xr]s^{-1}} \bmod p) \bmod q \\
&= (g^{[H(m)+xr][H(m)+xr]^{-1}k} \bmod p) \bmod q \\
&= (g^k \bmod p) \bmod q = r
\end{aligned}$$

上面证明中用到以下结果。

若 $m=n(\bmod q)\rightarrow g^m=g^n(\bmod p)$，其中 g 的阶为 q。

在上述 DSA 的验证算法中，使用了 $s^{-1}\bmod q$，这就要求 $s\neq 0(\bmod q)$。这一点可通过下述办法解决：在 DSA 算法中，如果 $s\equiv 0(\bmod q)$，那么签名者重新选择一个 x 构造一个新的签名，这样做不会产生什么问题，因为在实际应用中，出现 $s\equiv 0(\bmod q)$ 的概率很小，大约为 2^{-160}，所以可认为这种情况很难发生。

当 p 选为 512 比特时，ElGamal 签名的长度为 1024 比特，而在 DSA 中通过一个 160 比特的素数 q 将签名的长度降低为 320 比特，这就大大减少了存储空间和传输带宽。

DSA 数字签名算法流程如图 11.1 所示。

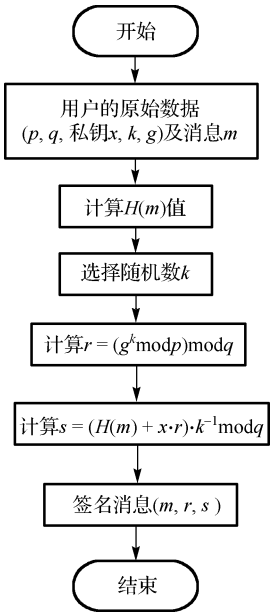

图 11.1　DSA 数字签名算法流程图

11.2 DSA 数字签名算法相关模块 FPGA 设计

根据上面的原理描述，可知实现该签名算法主要需要五个模块，DSA 数字签名算法的 FPGA 实现的关键模块如图 11.2 所示。

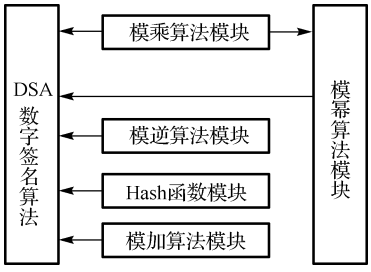

图 11.2　DSA 数字签名算法的关键模块

11.2.1　模乘算法模块设计

素数域 F_p 上的乘法 $a \cdot b$ 可以通过先将 a 和 b 作整数乘法，然后以 p 为模做取模运算来完成。Montgomery 方法的思想是在普通的约减算法中用简单的运算代替除法运算，这一方法对单个模乘运算效率并不高，但是用它计算模幂却非常有效，因为计算模幂时要完成给定输入的多次模乘。Montgomery 模乘能将模 n 运算转换为模 2^k，而模 2^k 能够通过取数的低 k 位来实现，因而避免繁琐的模 n 除法运算。

下面再简单回顾一下 Montgomery 算法的基本思想。

设 N 为 k 比特的整数，即：$2^{k-1} < N < 2^k$，$A < N$，$B < N$；

$\gcd(N,r)=1$，N 和 r 互素，取 $r=2^k$；r^{-1} 是 r 模 N 的逆元，即：$r * r^{-1} = 1 \bmod N$。

计算 $\mathrm{Mon}P(A,B,r,N) = A * B * r^{-1} (\bmod N)$ 的算法如下。

$\mathrm{Mon}P(A,B,r,N)$

{

t=A * B;

$u = (t + (t * N' \bmod r)N) / r$;

if $u \geqslant N$

return $u - N$;

else return u.

}

因为 $r=2^k$，所以对 r 求模以及对 r 整除的运算都可以通过简单的移位操作来完成，避免了复杂的传统除法运算。因为 Montgomery 算法计算的是 $A * B * \mathrm{r}^{-1} (\bmod N)$ 的值，并不是 $A * B (\bmod N)$ 的值，所以还需要进行预计算或者后处理以消除 r^{-1} 的影响。

模乘算法模块的程序实现，参考程序 11-1。

程序 11-1：

```
    assign product = prodreg4[MPWID - 1:0];
    assign prodreg1 = (mpreg[0] == 1'b1) ? prodreg + mcreg : prodreg;
    assign prodreg2 = prodreg1 - modreg1;
    assign prodreg3 = prodreg1 - modreg2;
    assign modstate = {prodreg3[MPWID + 1], prodreg2[MPWID + 1]};
    assign prodreg4 = (modstate == 2'b11) ? prodreg1 :  (modstate == 2'b10) ?
                      prodreg2 : prodreg3;
    assign mcreg1 = mcreg - modreg1;
    assign mcreg2 = (mcreg1[MPWID] == 1'b1) ? mcreg :  mcreg1;
    assign ready = first;
    always @(posedge clk or first or ds or mpreg or posedge reset)
    begin:
      if (reset == 1'b1)
         first <= 1'b1;
      else
      begin
         if (first == 1'b1)
         begin
```

```
            if (ds == 1'b1)
            begin
               mpreg <= mplier;
               mcreg <= {2'b00, mpand};
               modreg1 <= {2'b00, modulus};
               modreg2 <= {1'b0, modulus, 1'b0};
               prodreg <= {MPWID+2{1'b0}};
               first <= 1'b0;
            end
         end
         else
            if (mpreg == 0)
               first <= 1'b1;
            else
            begin
               mcreg <= {mcreg2[MPWID:0], 1'b0};
               mpreg <= {1'b0, mpreg[MPWID - 1:1]};
               prodreg <= prodreg4;
            end
      end
   end
```

在程序 11-1 中，定义一个整型变量 MPWID 作为此模块的宽度，当用此模块时，修改此参数，可以达到控制宽度的目的。乘数和被乘数必须是一个小于模量的值。

11.2.2 模幂算法模块设计

模幂算法 $c = m^e \bmod n$ 就是进行一系列的模乘运算， m 为 i 位二进制整数 $(m_{i-1}, m_{i-2}, \cdots, m_0)_2$，即待加密的数据； n 为 i 位二进制整数 $(n_{i-1}, n_{i-2}, \cdots, n_0)_2$，即模数； e 为 i 位二进制整数 $(e_{i-1}, e_{i-2}, \cdots, e_0)_2$，为加密密钥。根据扫描密钥的方向不同，模幂算法可以分为两种，一种为从左到右的扫描，一种为从右到左的扫描。 两种方法的执行时间不一样，前者采用串行计算，后者采用并行计算，平均条件下，后者比前者快 1.5 倍，但它的硬件规模要大一些。为了提高速度，本实例采用第二种方法，其算法如下。

$me(m, e, n)$
{
  $c = 1;$
  $p = m \bmod n;$
  for $j = 1$ to $i - 1$ do
    {
      if $(e[j] == 1)$ $c = p \cdot c \bmod n;$
      $p = p \cdot p \bmod n;$
    }
  return $c;$
}

在算法的循环体中，运算 $c = p \cdot c \bmod n$ 和 $p = p \cdot p \bmod n$ 可以由两个模乘器并行完成。由此可见，完成一次模幂运算需要进行 $i+1$ 次模乘运算。模幂算法模块的程序实现，参考程序 11-2。

程序 11-2：

```
assign ready = done;
assign bothrdy = multrdy & sqrrdy;
modmult #(KEYSIZE) modmultiply(.mpand(tempin), .mplier(sqrin),
            .modulus(modreg), .product(tempout), .clk(clk),
            .ds(multgo), .reset(reset), .ready(multrdy));
modmult #(KEYSIZE) modsqr(.mpand(root), .mplier(root), .modulus(modreg),
            .product(square), .clk(clk), .ds(multgo), .reset(reset),
            .ready(sqrrdy));
always @(posedge clk or posedge reset or done or ds or count or bothrdy)
 begin: mngcount

     if (reset == 1'b1)
     begin
        count <= {KEYSIZE{1'b0}};
        done <= 1'b1;
     end
     else
     begin
        if (done == 1'b1)
        begin
           if (ds == 1'b1)
           begin
              count <= {1'b0, inExp[KEYSIZE - 1:1]};
              done <= 1'b0;
           end
        end
        else if (count == 0)
        begin
           if (bothrdy == 1'b1 & multgo == 1'b0)
           begin
              cypher <= tempout;
              done <= 1'b1;
           end
        end
        else if (bothrdy == 1'b1)
        begin
           if (multgo == 1'b0)
              count <= {1'b0, count[KEYSIZE - 1:1]};
        end
     end
```

```
 end
always @(posedge clk or posedge reset or done or ds)
 begin: setupsqr

   if (reset == 1'b1)
   begin
      root <= {KEYSIZE{1'b0}};
      modreg <= {KEYSIZE{1'b0}};
   end
   else
   begin
      if (done == 1'b1)
      begin
         if (ds == 1'b1)
         begin
            modreg <= inMod;
            root <= indata;
         end
      end
      else
         root <= square;
   end
 end
always @(posedge clk or posedge reset or done or ds)
 begin: setupmult
   if (reset == 1'b1)
   begin
      tempin <= {KEYSIZE{1'b0}};
      sqrin <= {KEYSIZE{1'b0}};
      modreg <= {KEYSIZE{1'b0}};
   end
   else
   begin
      if (done == 1'b1)
      begin
         if (ds == 1'b1)
         begin
            if (inExp[0] == 1'b1)
               tempin <= indata;
            else
            begin
               tempin[KEYSIZE - 1:1] <= {31{1'b0}};
               tempin[0] <= 1'b1;
            end
```

```
                modreg <= inMod;
                sqrin[KEYSIZE - 1:1] <= {31{1'b0}};
                sqrin[0] <= 1'b1;
             end
          end
          else
          begin
             tempin <= tempout;
             if (count[0] == 1'b1)
                sqrin <= square;
             else
             begin
                sqrin[KEYSIZE - 1:1] <= {31{1'b0}};
                sqrin[0] <= 1'b1;
             end
          end
       end
    end

    always @(posedge clk or posedge reset or done or ds or count or bothrdy)
    begin: crypto

       if (reset == 1'b1)
          multgo <= 1'b0;
       else
       begin
          if (done == 1'b1)
          begin
             if (ds == 1'b1)
                multgo <= 1'b1;
          end
          else if (count != 0)
          begin
             if (bothrdy == 1'b1)
                multgo <= 1'b1;
          end
          if (multgo == 1'b1)
             multgo <= 1'b0;
       end
    end
```

在程序 11-2 中，通过两次调用模乘算法模块，分别实现相乘与平方的功能，然后通过对模乘与平方的循环操作，可以实现模幂算法功能。其中 KEYSIZE 控制模乘和模幂的数据宽度。

11.2.3 模逆算法模块设计

通常素数域上的求逆算法有扩展的欧几里德算法和 Montgomery 求逆算法，扩展的欧几里德算法可以用来求两个数的最大公约数，也可以求模逆。算法的原理是辗转相除法，因此过程中用到了很多的除法和减法操作，不利于硬件的实现。

本实例从降低算法层次和减少触发操作的角度出发，对其进行了改进，改进的算法如下。

输入：模 p，整数 a，求 a 的逆元；

输出：$a^{-1}\bmod p$。

第一步 $(x,u)\leftarrow(1,a)$；

第二步 $(y,v)\leftarrow(0,p)$；

第三步 当 $u\neq 1$ 和 $v\neq 1$ 时，重复执行下面操作

当 $u_0=0$，重复执行 $u=u\gg 1$；

若 $x_0=0$，$x=x\gg 1$；

否则，$x=x+p\gg 1$；

当 $v_0=0$，重复执行 $v=v\gg 1$；

若 $y_0=0$，$y=y\gg 1$；

否则，$y=y+p\gg 1$

当 $u\geqslant v$，$(x,u)=(x-y,u-v)$；

否则，$(y,v)=(y-x,v-u)$；

当 $u=1$，则返回 $x=a^{-1}\bmod p$，否则返回 $y=a^{-1}\bmod p$。

算法中使用公约数的性质，将扩展欧几里德算法中的除法运算全部改成了减法，指令时间比除法操作要短；另外，将二进制中的除法运算运用到素数域里，并且经过改进能将素数域和二元域的求模逆运算结合起来，这样在硬件实现时，可以使用相同的控制单元，有利于节省成本。模逆算法模块的程序实现，参考程序 11-3。

程序 11-3：

```
always @(posedge clk)
  begin
      if (en == 1'b0)
         begin
            b <= {SIZE{1'b0}};
            u <= a;
            v <= p;
                  s <= SIZE 'h1;
            r <= { SIZE {1'b0}};
            st <= s0;
            rdy <= 1'b0;
          end
       else
         begin
              case (st)
```

```
s0 :
  if(u/2==0)
          begin
             st <= s0;
       u <= u/2;
                    if(s/2 == 0)
                      s <= s/2;
                      else
                        s <= (s+p)/2;
                    end
          else
            st <= s1;
s1 :
        if(v/2==0)
          begin
             st <= s1;
       v <= v/2;
                        if(v/2 == 0)
                      r <= r/2;
                      else
                        r <= (r+p)/2;
                end
        else
              st <= s2;
s2 :
     if(u>=v)
               begin
          u <= u-v;
          s <= s-r;
                         st <= s3;
           end
            else
               begin
          v <= v-u;
          r <= r-s;
                         st <= s3;
         end
s3 :
           if(u!=1 && v!=1)
               st <=s0;
     else if(u==1)
                  begin
                     b <=((s + p)>p)?s:(s + p);
                      rdy <=1'b1;
```

```
                                    end
                          else
                                        begin
                                         b <=((r + p)>p)?r:(r + p);
                                         rdy <=1'b1;
                                         end
                             default :
                        begin
                                   b <=0;
                         st <= s0;
                         end
                endcase
            end
        end
```

在程序 11-3 中，使用 case 语句来实现上述改进模逆算法中的第三步，在 case 语句中嵌套 if 语句来判断该条 case 语句是否结束。

11.2.4 模加算法模块设计

素域上的加减法和通常代数学中的整数加减法基本相同，唯一不同是由于计算结果必须保证是素域 F_p 中的元素，所以对最后的结果要增加一步“求模 p 的剩余”的运算，就是通常所说的取模运算。

下面给出了素域 F_p 上加法的具体算法。

输入：模数 p 和整数 a，$b \in [0, p-1]$。

输出：$c = (a + b) \bmod p$。

（1）$c = a + b$；

（2）若 $c \geqslant p$，则 $c = c - p$；

（3）返回 c。

模加算法模块的程序实现，参考程序 11-4。

程序 11-4：

```
    always @(posedge clk)
    begin
        if (en == 1'b0)
           begin
              c <= {511{1'b0}};
              d <=512'h0;
              st <= s0;
              rdy <= 1'b0;
            end
         else
           begin
```

```
                case (st)
                  s0 :
                    begin
                       d <= a+b;
                          st <= s1;
                          end
                     s1 :
                         begin
                          if(d>p)
                             begin
                            d<= d-p;
                                st <= s1;
                           end
                            else
                              begin
                              rdy <= 1'b1;
                           d<= c;
                               st <= s1;
                          end
                       default :
                          begin
                            st <= s0;
                            rdy <= 1'b0;
                           end
                     endcase
            end
    end
```

在程序 11-4 中，在 s0 时间段，实现了两个大素数的相加，结果存在中间变量 *d* 中；在 s1 时间段，先判断 *d* 与模数 *p* 的大小，若 *d*>*p*，将 *d*–*p* 的结果赋值于 *d*，否则 *d* 值不变。最后将 *d* 的值赋给 *c*，*c* 即为最终结果。

11.3 DSA 数字签名算法的工程实现及结果

1．创建 ISE 工程

打开 ISE 开发环境，选择菜单 File→New Project，建立一个新工程，然后设置顶层文件模块名、存储目录和设计方式。再选择器件，这里选择 Xilinx 的 Spartan6 来实现，最后选择第三方仿真软件 ModelSim 和 Verilog 语言进行仿真测试。新工程的建立如图 11.3 所示。

2．编写设计代码

在 11.2 节中详细叙述了 DSA 算法的关键 FPGA 模块设计思路，这里将上述模块加载到 ISE 中，模块具体如图 11.3 至图 11.5 所示。

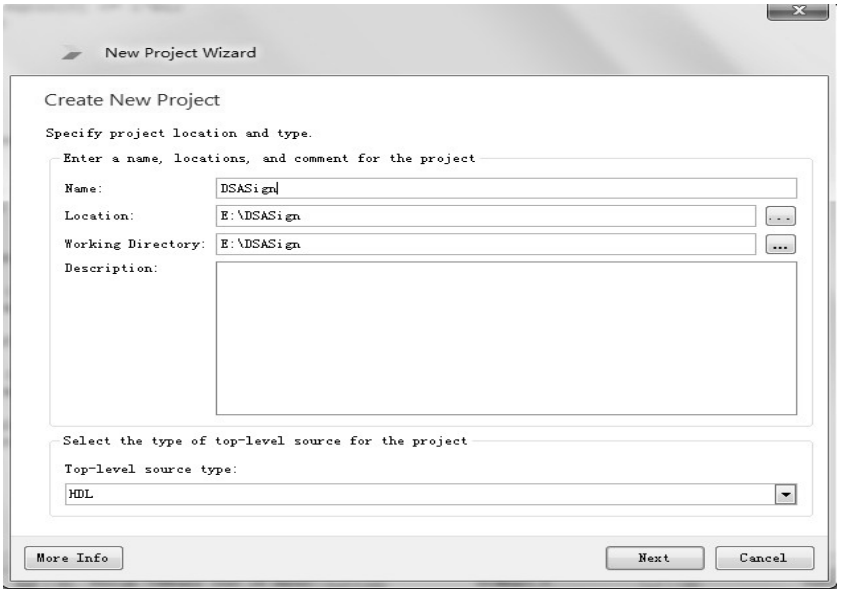

图 11.3 DSA 数字签名算法 FPGA 工程的建立

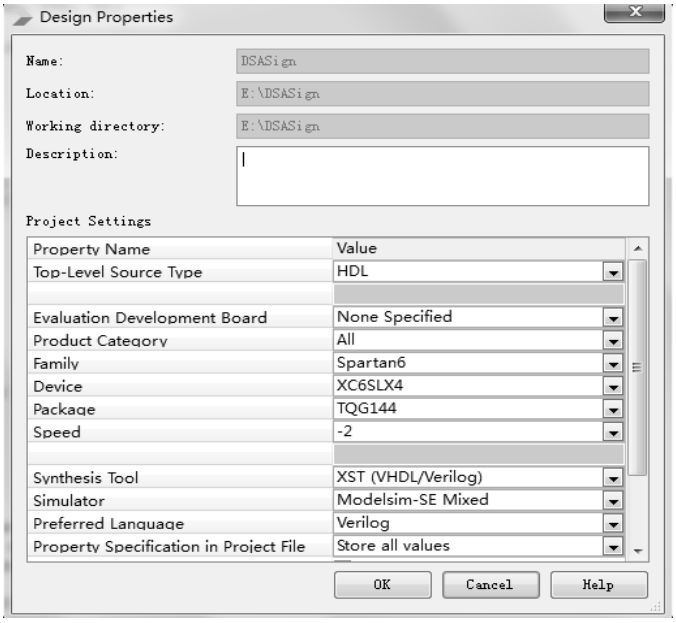

图 11.4 算法工程设置

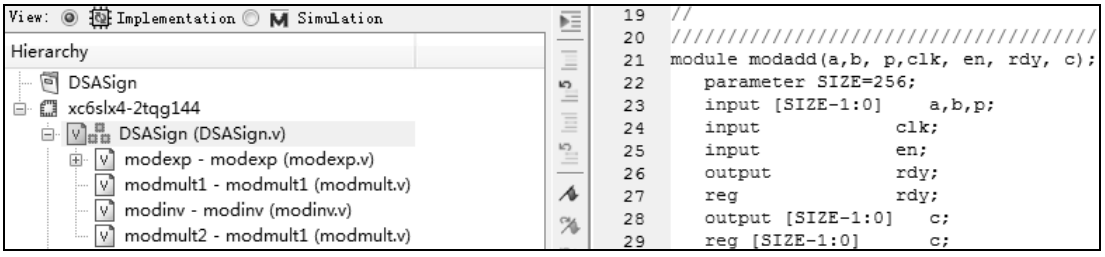

图 11.5 DSA 数字签名算法的实现

3．工程验证

（1）根据以上编写的程序，编写该程序的测试文件，如图 11.6 和图 11.7 所示。

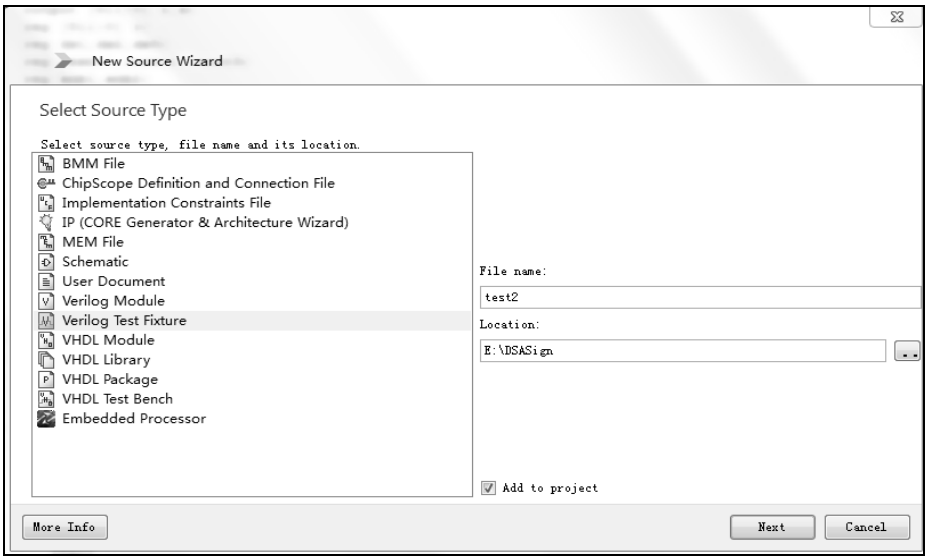

图 11.6　DSA 数字签名算法测试文件的建立

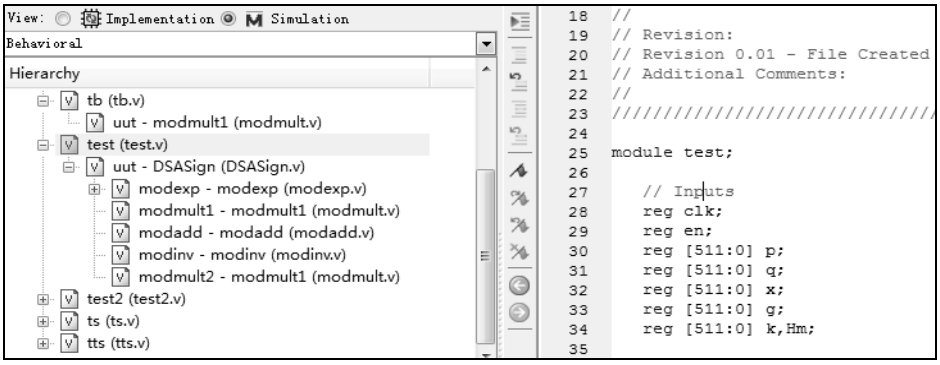

图 11.7　测试程序

（2）根据上一步的测试文件，调用 ModelSim 仿真软件，如图 11.8 所示，对该算法进行仿真测试，仿真结果如图 11.9 所示。

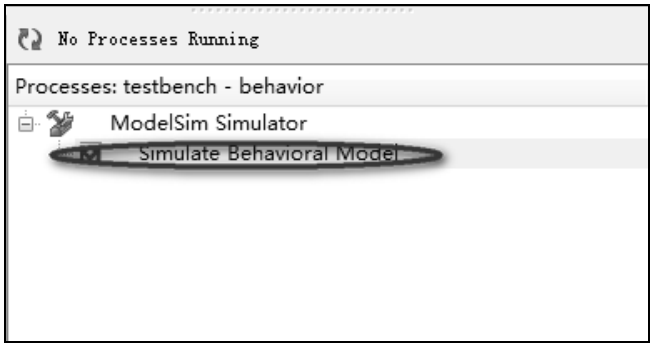

图 11.8　调用 ModelSim 仿真软件

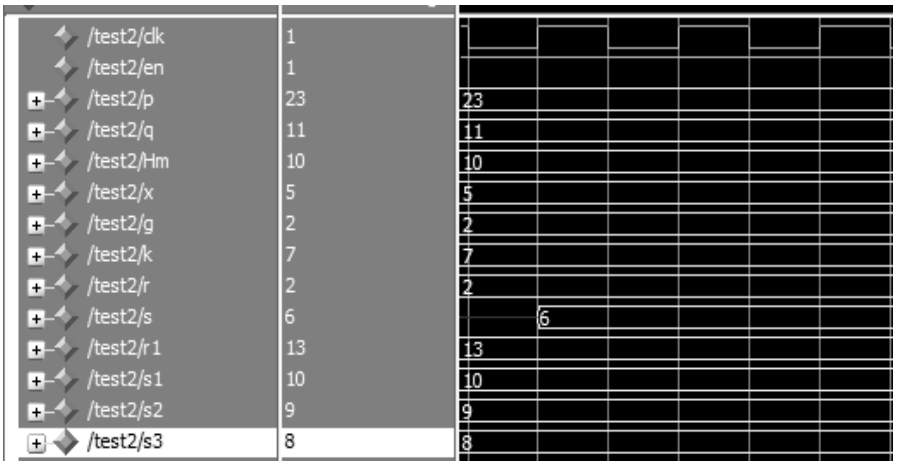

图 11.9 DSA 数字签名算法 FPGA 实现仿真图

11.4 效果测试

为了验证设计的正确性，本实例采用 ISE 14.3 开发工具对系统设计进行了整合和综合，编写了 test bench，用 ModelSim SE10.1 仿真工具进行了仿真，包括各个子模块的仿真和整个系统的仿真，软件仿真通过后又将综合后的编程文件烧录到 FPGA 中进行验证。该测试所使用的计算机配置为：四核 i5-3470 CPU，4G 内存，32 位操作系统。

由于每个用户拥有的 DSA 公私钥都不相同，所以本实例所设计的硬件部分并不包括公私钥对的生成部分，公私钥对的生成部分另外通过软件部分来实现。

为了便于观察实验结果，设计了一组小数用于数字签名的最终实现。选取参数如下所示。

Hm=512'd10；

p = 512'd23；

q = 512'd11；

x = 512'd5；

g = 512'd2；

k = 512'd7；

结果为：

r=512'h2；

s=512'h6。

其中，Hm 是经过 SM3 算法 Hash 运算后的结果，x 为私钥。

下面对该工程的时序、运行效率、资源占用情况进行分析。在 Xilinx 公司的 6slx4tqg144 FPGA 元器件上，经过 ISE 的 XST 分析工具进行综合。综合结果如图 11.10 所示。

资源占用情况分析报告如表 11.1 所示。

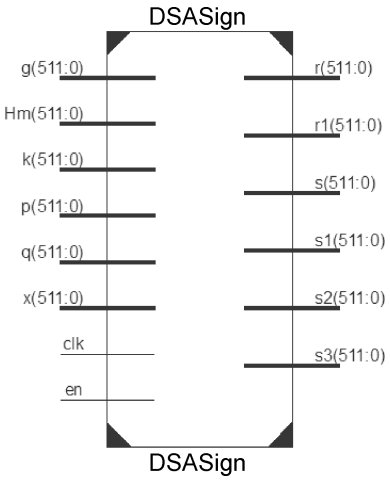

图 11.10　DSA 数字签名算法外部引脚

表 11.1　DSA 数字签名算法资源占用情况

Selected Device : 6slx4tqg144-2	
Number with an unused Flip Flop	23166　out of　38595　60%
Number with an unused LUT	4891　out of　38595　12%
Number of fully used LUT-FF pairs	10538　out of　38595　27%
Number of BUFG/BUFGCTRLs	1　out of　16　6%

时序分析报告如下。

Timing Summary：

Minimum period: 19.336 ns (Maximum Frequency: 51.716 MHz)

Minimum input arrival time before clock: 20.581 ns

Maximum output required time after clock: 22.372 ns

Maximum combinational path delay: 23.235 ns

由时序分析报告结果可知，DSA 数字签名算法的最高工作频率为 51.716 MHz。由仿真结果可知，每处理 512 比特的数据需要经过 5760 个时钟。所以，DSA 数字签名算法的最高处理速度计算结果为 512×51.716/5760=4.6 Mbps。

11.5　本章小结

DSA 是 Schnorr 和 ElGamal 签名算法的变种，被美国 NIST 作为数字签名标准（Digital Signature Standard，DSS）。DSA 算法的安全性基于整数有限域离散对数难题，其安全性与 RSA 相比差不多。而与 RSA 算法不同之处在于，它不能用作加密和解密，也不能进行密钥交换，它只用于签名和验签，速度比 RSA 算法要快很多。

本章首先对 DSA 算法的原理、工作模式与安全性进行了介绍，之后重点介绍了 DSA 算法的 FPGA 实现方法，包括模乘算法、模幂算法、模加算法和模逆算法的设计实现，给出了硬件实现方案和相关代码，最后给出了仿真结果和综合结果，并对结果进行了分析，使读者不仅能从理论上了解 DSA 算法，更能从算法的 FPGA 实现过程中加深对算法的理解。

第 12 章　ECC 数字签名算法 FPGA 实现

12.1 ECC 数字签名原理

ECC 数字签名算法本质是 ECC 公钥加密算法的逆运算。签名时，发送方用自己的私钥对消息进行加密，接收方收到消息后用发送方的公钥进行解密，由于私钥由发送方自己保管且只有他本人知道，如果只知道发送方的公钥，不可能伪造其签名，从而起到发送方不可抵赖的效果。公证的第三方可以用发送方的公钥对签名的消息进行解密，从而验证消息确实来自于该发送方。

设 E 为定义在有限域 F_p 上的椭圆曲线，以 $E(F_p)$ 为椭圆曲线 E 在 F_p 中的有理集，它是一个有限群。在 $E(F_p)$ 中选一个点 G，称为基点（Base Point），设 G 的阶（Order）为 n（通常要求 n 是一个大素数）。每个用户选取一个整数 k（$1 < k < n$）作为私钥，而点 $P = kG$ 作为其公钥，这样便形成一个椭圆曲线密码系统（ECC）。该系统的公开参数有：有限域 F_p（p 为大于 3 的素数），椭圆曲线 E（比如 $p > 3$ 时，由 $y^2 = x^3 + ax + b$，$4a^3 + 27b^3 \neq 0$ 定义的曲线），基点 G 及其阶 n，每个用户的公钥 $P = kG$（私钥 k 严格保密）。实现过程还需要安全的 Hash 函数（比如我国的 SM3 杂凑算法）。

ECC 数字签名流程图如图 12.1 所示，实现过程中主要用到的算法是椭圆曲线数字签名算法（Elliptic Curve Digital Signature Algorithm，ECDSA），以用户 A 向用户 B 发送签名消息为例，用户 A 要完成的签名算法如下。

第一步，用户 A 选择一个随机或者伪随机数 $k(1 < k < n-1)$；

第二步，用户 A 计算 $kG = (x_1, y_1)$；

第三步，用户 A 计算 $r = h(m) + x_1 \bmod n$　（其中 m 为要签名的消息，h 为 Hash 函数）；

第四步，若 $r + d = 0 (\bmod n)$，返回到第一步，否则计算 $(r + d)^{-1} \bmod n$；

第五步，用户 A 计算 $s = (r + d)^{-1}(k - d * r) \bmod n$，若 $s = 0$，返回到第一步；

最后一步，用户 A 发送签名消息 (m, r, s) 给用户 B。

用户 B 接收到签名消息后，利用 ECDSA 验证算法，对签名消息验证如下。

第一步，用户 B 验证 r 和 s 是否是区间 $[1, n-1]$ 中的整数；

第二步，用户 B 计算 $e = h(m)$；

第三步，用户 B 计算 $X = r \cdot s \cdot G + (r + s) \cdot P_A$，设 $X = (x_2, y_2)$；

最后一步，若 $h(m) = r - x_2 (\bmod n)$，则接受签名，否则拒绝签名。

复杂度分析：签名算法里有 1 个点乘运算，1 个求逆和 2 个乘法运算。验证算法里有两个点乘运算，1 个点加运算，1 个乘法运算。当然，验证通常比签名慢。这一签名方案在一般随机模型下是可证安全的，验证运算也比较简单。

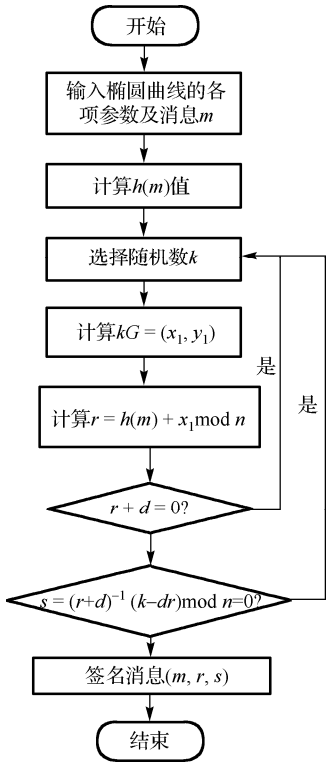

图 12.1　ECC 数字签名流程图

12.2　ECC 数字签名算法相关模块 FPGA 设计

通过对算法结构的分析发现，实现该签名算法主要需要九个模块：数字签名模块、点乘模块、Hash 模块、模加模块、模乘模块、模逆模块、点加模块、2 倍点模块和模方模块。ECC 数字签名算法的关键模块如图 12.2 所示。

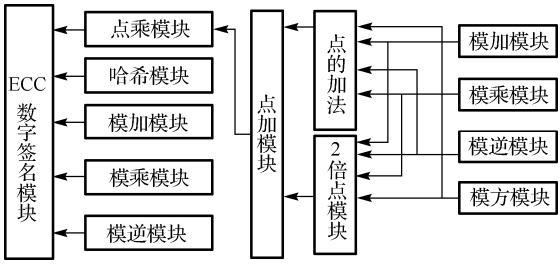

图 12.2　ECC 数字签名的关键模块

12.2.1　模乘算法模块设计

在多项式基表示下，有限域 GF(2^m)的乘法可以看成模不可约多项式 $f(x)=x^m+r(x)$ 意义下的多项式乘法。这个乘法过程包括了多项式的相乘和求余两个过程，除了基本的与、或、异或以外，有限域 GF(2^m)上的乘法没有其他的 CPU 指令可以利用。因此，实现多项式相乘和

求余的基本方法就是移位相加方法。下面讨论在通用计算机上实现移位相加的有限域乘法。

设计算机的字长为 w，记 $s=\lceil m/w \rceil$，$t=w\cdot s-m$。对于任意一个域元素 $a=(a_{m-1}\cdots a_1 a_0)$，$a\in \mathrm{GF}(2^m)$，把它存储在一个长为 s 的字数组中：$A=(A[s-1],\cdots,A[1],A[0])$，其中 $A[0]$的最右位是 a_0，$A[s-1]$的左边 t 个位均为 0。

域中乘法的移位相加算法有如下的常见形式，该算法是基于下列计算方式进行的：

$$a\cdot b=a_{m-1}x^{m-1}b+\cdots+a_2x^2+a_1xb+a_0b$$

实现多项式模乘的最基本的方法是移位相加法，具体算法如下。

输入：$A(x)=a_{n-1}x^{n-1}+\cdots+a_1x+a_0$

$B(x)=b_{n-1}x^{n-1}+\cdots+b_1x+b_0$

输出：$C(x)=A(x)\cdot B(x) \bmod f(x)$

（1）若 $a_0=1$，则有 $C=B$；

（2）对 i 从 1 到 $n-1$，执行

$B=(B\ll 1) \bmod f(x)$；

若 $a_i=1$，则 $C=C$ xor B；

（3）返回 C。

该算法需要进行大量的移位操作，虽然不适合基于处理器的实现方案，但是对于硬件实现来说是很方便的，而且不需要占用任何逻辑单元，所以在用 FPGA 进行硬件实现时还是采用这种移位操作的方法。

模乘算法模块的程序实现，参考程序 12-1。

程序 12-1：

```
always @(posedge clk)
begin:
 if (en == 1'b0)
   begin
      ct = {8{1'b0}};
      av = {m{1'b0}};
      as = {k{1'b0}};
      cv = {m{1'b0}};
      state <= s_state_s0;
      rdy <= 1'b0;
   end
   else
      case (state)
         s_state_s0 :
            begin
               if (ct == 0)
                  av = a;
                  as = av[m - 1:m - k];
                  for (i = k - 1; i >= 0; i = i - 1)
                   begin
```

```
                if (as[i] == 1'b1)
                   cv = cv ^ b;
                   cm = cv[m - 1];
                   cv = {cv[m - 2:0], 1'b0};
                if (cm == 1'b1)
                   begin
                    cv[74] = cv[74] ^ 1'b1;
                    cv[0] = 1'b1;
                   end
               end
            if (ct < n - 1)
               begin
                ct = ct + 1;
               state <= s_state_s0;
               end
            else
              state <= s_state_s1;
            av = {av[m - k - 1:0], ak};
         end
      s_state_s1 :
         begin
            as = av[m - 1:m - k];
            if (r > 1 & k > 1)
            begin
              for (i = k - 1; i >= k - r + 1; i = i - 1)
              begin
                if (as[i] == 1'b1)
                   cv = cv ^ b;
                cm = cv[m - 1];
                cv = {cv[m - 2:0], 1'b0};
                if (cm == 1'b1)
                begin
                   cv[74] = cv[74] ^ 1'b1;
                   cv[0] = 1'b1;
                end
              end
              if (as[k - r] == 1'b1)
                cv = b ^ cv;
            end
            else
              if (as[k - 1] == 1'b1)
                cv = b ^ cv;
            c <= cv;
            rdy <= 1'b1;
            state <= s_state_s2;
```

```
                end
            s_state_s2 :
                state <= s_state_s2;
              default :
                begin
                   state <= s_state_s0;
                   rdy <= 1'b0;
                end
        endcase
    end
endmodule
```

在程序 12-1 中，在时钟上升沿时段，en 复位键低位有效，然后设置成高位有效，通过对状态值 state 进行控制运算，达到实现模乘算法的目的。

12.2.2 模逆模块设计

在 ECC 算法实现中，一个重要的运算是两个多项式的除法，除法操作可以通过乘以乘数逆元来实现。模逆算法的快速实现如下所示。

输入：A 以及模数 n

输出：$A^{-1}\bmod n$

（1）$T \leftarrow A^2$

（2）$X \leftarrow AT$

（3）$T \leftarrow X^2$

（4）$X \leftarrow AT$

（5）$T \leftarrow X^2$

（6）连续执行 2 次 $T \leftarrow T^2$

（7）$X \leftarrow XT$

（8）$T \leftarrow X^2$

（9）$X \leftarrow AT$

（10）$T \leftarrow X^2$

（11）连续执行 6 次 $T \leftarrow T^2$

（12）$X \leftarrow XT$

（13）$T \leftarrow X^2$

（14）连续执行 13 次 $T \leftarrow T^2$

（15）$X \leftarrow XT$

（16）$T \leftarrow X^2$

（17）$X \leftarrow AT$

（18）$T \leftarrow X^2$

（19）连续执行 28 次 $T \leftarrow T^2$

（20）$X \leftarrow XT$

（21）$T \leftarrow X^2$

（22）连续执行 57 次$T \leftarrow T^2$

（23）$X \leftarrow XT$

（24）$T \leftarrow X^2$

（25）连续执行 115 次$T \leftarrow T^2$

（26）$X \leftarrow XT$

（27）$A \leftarrow T^2$

（28）返回A

程序 12-2：

```
always @(posedge clk)
begin
   if (en == 1'b0)
     begin
        aout <= {233{1'b0}};
        cnt1 <= {3{1'b0}};
        cnt2 <= {3{1'b0}};
        cnt3 <= {4{1'b0}};
        cnt4 <= {5{1'b0}};
        cnt5 <= {6{1'b0}};
        cnt6 <= {7{1'b0}};
        x <= {233{1'b0}};
        t <= {233{1'b0}};
        st <= s0;
        rdy <= 1'b0;
        enb1 <= 1'b0;
        enb2 <= 1'b0;
       end
    else  begin
          case (st)
            s0 :
              begin
                st <= s1;
                t <= m1;
              end
            s1 :
                   if(ready1 == 1)
                     begin
                st <= s2;
                x <= n1;
                       enb1 <= 0;
                     end
                   else
                     begin
```

```
            enb1 <= 1;
          st <= s1;
       end
……
 s5 :
        if(cnt1 < 2)
            begin
       t <= m3;
       st <= s5;
       cnt1 <=cnt1 + 1'b1;
    end
        else
          st <= s6;
       ……
        s10 :
          if(cnt2 < 6)
             begin
       t <= m3;
       st <= s10;
       cnt2 <=cnt2 + 1'b1;
     end
         else
           st <= s11;
      …
 s13 :
      if(cnt3 < 13)
            begin
           t <= m3;
          st <= s13;
          cnt3 <=cnt3 + 1'b1;
        end
          else
             st <= s14;
               …
 s18 :
      if(cnt4 < 28)
            begin
           t <= m3;
         st <= s18;
         cnt4 <=cnt4 + 1'b1;
      end
          else
            st <= s19;
               …
```

```
        s21 :
          if(cnt5 < 57)
                  begin
            t <= m3;
            st <= s21;
            cnt5 <=cnt5 + 1'b1;
           end
               else
                 st <= s22;
                …
        s24 :
          if(cnt6 < 115)
                  begin
            t <= m3;
            st <= s24;
            cnt6 <=cnt6 + 1'b1;
           end
               else
                 st <= s25;
       s25 :
          if(ready2 == 1)
                 begin
              st <= s26;
              x <= n2;
                           enb2 <= 0;
           end
              else
                  begin
                   enb2 <= 1;
             st <= s25;
           end
        s26 :
          begin
               st <= s26;
               aout <= m2;
               rdy <= 1'b1;
          end
        default :
          begin
            st <= s0;
            rdy <= 1'b0;
          end
      endcase
    end
```

```
        end
    mult_ip inst1(a, t, clk, enb1, ready1, n1);
    mult_ip inst2(x, t, clk, enb2, ready2, n2);
    squar pingfang1(a,m1);
    squar pingfang2(x,m2);
    squar pingfang3(t,m3);
    endmodule
```

在程序 12-2 中，模平方的次数为 232 次，模乘的次数为 10 次。在该程序中分别用 cnt1，cnt2，cnt3，cnt4，cnt5，cnt6 来控制其中的循环次数。

12.2.3 Hash 函数模块设计

ECC 数字签名算法中需要用到 Hash 函数运算，本书采用的是我国的商密算法 SM3 算法。关于 SM3 算法的相关原理与实现，在本书中给出了详细的介绍与实现，请参考本书第 10 章。另外，取 SM3 算法结果的低 233 比特作为本章 Hash 的最终输出结果。

12.2.4 点乘模块设计

点乘又称多倍点乘法或标量乘法，其定义如下：

$$Q = KP = P + P + \cdots + P,\ \ K \in \mathrm{GF}(2^m)$$

本书采用的是 LD 投影坐标下的 Montgomery 方法实现点乘运算，该方法具体为对 K 进行从高位至低位的逐位扫描，如果 k_i =1 或 0，每次均先执行一次点加后再执行一次倍点，唯一不同的是源与目的操作数不同。由此使得每比特对应的运算时间均为定值，具体操作也固定，这将能帮助硬件有效抵御功耗分析攻击；如果能够在编码时确保 K 的最高位（第 233 位）为 1，则每次 KP 运算的总时间也是固定的。这样的特性将非常有利于硬件抵御时间分析攻击，大大提高硬件的设计安全性。

根据控制电路给出的状态，消息输出模块按照算法要求的步骤完成消息的输出。具体过程为：控制状态机通过读状态控制键 read_counter 来实现操作，复位键完成对 read_counter 的初始化，分 8 段完成 Hash 值的输出，这是因为本实例设计的数据端口为 32bit 宽，这样设计的初衷是便于将设计的模块挂载到目前常用的数据总线上，数据是按照每个时钟周期 32bit 进入 Hash 单元。

LD 投影坐标下的 Montgomery 点乘算法

输入：任一正整数 $K = (k_{m-1}, k_{m-2}, \cdots, k_0)$，其中 $k_{m-1} = 1$，任一在给定椭圆曲线上的点 P，P 的仿射坐标为 (x_0, y_0)。（注：这里给定椭圆曲线为 $y^2 + xy = x^3 + 1$）

输出：点 $Q = KP$ 的仿射坐标 (x_3, y_3)。

（1）如果 $K = 0$ 或 $x_0 = 0$，则跳出运算，直接输出 $(0,0)$ 至 Q 点。

（2）$X_1 \leftarrow x_0, Z_1 \leftarrow 1, X_2 \leftarrow x_0^4 + 1, Z_2 \leftarrow x_0^2$。

（3）对于 i 从 m–2 到 0，重复执行

若 $k_i = 1$，则依次执行

$$(X_1, Z_1) \leftarrow \mathrm{PADD}(X_1, Z_1, X_2, Z_2), (X_2, Z_2) \leftarrow \mathrm{PDOUBLE}(X_2, Z_2)。$$

若 $k_i=0$，则依次执行

$$(X_2,Z_2)\leftarrow \mathrm{PADD}(X_2,Z_2,X_1,Z_1),(X_1,Z_1)\leftarrow \mathrm{PDOUBLE}(X_1,Z_1)。$$

（4）$Q=\mathrm{PXY}(X_1,Z_1,X_2,Z_2)$。

（5）返回 $Q=(x_3,y_3)$。

其中，PADD 代表点加操作，PDOUBLE 代表倍点操作，PXY 代表坐标反变换一系列操作。

PADD(X_1,Z_1,X_2,Z_2) 算法

输入：在上述算法中的 $P=(x_0,y_0)$，点 P_1,P_2 的 LD 投影 x,z 坐标为

$$P_1(X_1,Z_1),P_2(X_2,Z_2)$$

输出：点 P_1+P_2 的 LD 投影 x,z 坐标 (X_1,Z_1)。

（1）$T_3=x_0$

（2）$X_1=X_1Z_1$

（3）$Z_1=Z_1X_2$

（4）$T_1=X_1Z_1$

（5）$Z_1=X_1+Z_1$

（6）$Z_1=Z_1^2$

（7）$X_1=Z_1T_3$

（8）$X_1=X_1+T_1$

（9）返回 (X_1,Z_1)

上述算法共需要 1 次模逆运算，10 次乘法运算，1 次模平方运算，6 次模加运算和 4 个中间存储变量。其中 T_1,T_3 为中间变量寄存器，X_1,Z_1 为寄存器型。

PDOUBLE(X_1,Z_1) 算法

输入：在上述算法中的 $P=(x_0,y_0)$，点 P_1 的 LD 投影 x,z 坐标为 (X_1,Z_1)

输出：点 $2P_1$ 的 LD 投影 x,z 坐标 (X_1,Z_1)。

（1）$X_1=X_1^2$

（2）$T_1=Z_1^2$

（3）$Z_1=T_1X_1$

（4）$T_1=T_1^2$

（5）$X_1=X_1^2$

（6）$X_1=X_1+T_1$

（7）返回 (X_1,Z_1)

上述算法共需要 1 次模乘运算，4 次模平方运算，1 次模加运算和 1 个中间存储变量。其中 T_1 为中间变量寄存器，X_1,Z_1 为寄存器型。

PXY(X_1,Z_1,X_2,Z_2) 算法

输入：在上述算法中的 $P=(x_0,y_0)$，点 P_1,P_2 的 LD 投影 x,z 坐标为

$$P_1(X_1,Z_1),P_2(X_2,Z_2)$$

输出：点 P_2 的仿射坐标 $(x_3,y_3)=(X_2,Z_2)$。

（1）如 $Z_1=0$，则跳出运算，直接输出(0,0)

（2）如 $Z_2 = 0$，则跳出运算，直接输出 $(x_0, x_0 + y_0)$

（3）$T_3 = x_0$

（4）$T_4 = y_0$

（5）$T_1 = Z_1 Z_2$

（6）$Z_1 = Z_1 T$

（7）$Z_2 = Z_2 T_3$

（8）$Z_1 = X_1 + Z_1$

（9）$X_1 = X_1 Z_2$

（10）$Z_2 = Z_2 + X_2$

（11）$Z_2 = Z_2 Z_1$

（12）$T_2 = T_3^2$

（13）$T_2 = T_2 + T_4$

（14）$T_2 = T_2 T_1$

（15）$T_1 = T_1 T_3$

（16）$T_2 = T_2 + Z_2$

（17）$T_1 = \text{inv}(T_1)$

（18）$T_2 = T_2 T_1$

（19）$X_2 = T_1 X_1$

（20）$Z_2 = X_2 + T_3$

（21）$Z_2 = Z_2 T_2$

（22）$Z_2 = Z_2 + T_4$

（23）返回 $(x_3, y_3) = (X_2, Z_2)$

上述算法共需要 1 次模逆运算，10 次乘法运算，1 次模平方运算，6 次模加运算和 4 个中间存储变量。其中 inv()表示模逆操作，其中 T_1, T_2, T_3, T_4 为中间变量寄存器，X_1, Z_1, X_2, Z_2 为寄存器型。这些模块的程序实现，参考如下所示。

程序 12-3：PADD 模块

```
always @(posedge clk)
  begin:
    begin
      if (en == 1'b0)
      begin
        X3 <= {233{1'b0}};
          Z3 <= {233{1'b0}};
        T1 = {233{1'b0}};
        T3 = {233{1'b0}};
        st <= s0;
        rdy <= 1'b0;
        enb1 <= 1'b0;
          enb2 <= 1'b0;
          enb3 <= 1'b0;
```

```
        enb4 <= 1'b0;
    end
    else
    begin
      case (st)
        s0 :
          begin
            st <= s1;
            T3 <= x0;
          end
        s1 :
             if(ready1 == 1)
              begin
             st <= s2;
             Z3 <= n1;
                enb1 <= 0;
              end
            else
              begin
                enb1 <= 1;
              st <= s1;
            end
        s2 :
           if(ready2 == 1)
              begin
            st <= s3;
            Z3 <= n1;
               enb2 <= 0;
              end
           else
             begin
               enb2 <= 1;
             st <= s2;
           end
        s3 :
          if(ready3 == 1)
           begin
            T1 <= n3;
            st <= s4;
               enb3 <= 0;
           end
            else
              begin
                enb3 <= 1;
```

```
                    st <= s3;
                  end
              s4 :
                begin
                    st <= s5;
                    Z3 <= m1;
                end
              s5 :
                begin
                  Z3 <= m2;
                  st <= s6;
                  cnt1 <=cnt1 + 1'b1;
                end
               s6 :
                if(ready4 == 1)
                    begin
                    st <= s7;
                    X3 <= n4;
                    enb4 <= 0;
                  end
                    else
                    begin
                      enb4 <= 1;
                    st <= s6;
                  end
               s7 :
                begin
                    st <= s7;
                    t <= m3;
                    rdy <= 1'b1;
                end
              default :
                begin
                  st <= s0;
                  rdy <= 1'b0;
                end
          endcase
        end
      end
    end

  mult_ip inst1(X1, Z2, clk, enb1, ready1, n1);
  mult_ip inst2(Z1, X2, clk, enb2, ready2, n2);
  mult_ip inst3(X1, Z3, clk, enb3, ready3, n3);
```

```
mult_ip inst4(Z3, T3, clk, enb4, ready4, n4);
modadd  add1(X3,Z3,m1);
modadd  add2(X3,Z3,m3);
squar pingfang3(Z3,m2);
endmodule
```

在程序 12-3 中，共利用 7 个状态循环，得到最终结果。利用一个 always 语句，通过对 st 的设置，达到输出点乘结果的目的。对不同的 st 值，调用不同的模块，实现上述所描的 PADD 算法。

程序 12-4：PDOUBLE 模块

```
always @(posedge clk)
  begin:
    begin
      if (en == 1'b0)
      begin
        X2 <= {233{1'b0}};
          Z2 <= {233{1'b0}};
        T1 = {233{1'b0}};
        st <= s0;
        rdy <= 1'b0;
        enb <= 1'b0;
      end
      else
      begin
        case (st)
          s0 :
            begin
              st <= s1;
              X2 <= m1;
                 T1 <= m2;
            end
          s1 :
               if(ready == 1)
                 begin
               st <= s2;
               Z2 <= n1;
                  enb <= 0;
                end
               else
                 begin
                   enb <= 1;
                 st <= s1;
              end
          s2 :
```

```
                    begin
                   st <= s3;
                   T1 <= m3;
                     end
              s3 :
                  begin
                   X2 <= m4;
                   st <= s4;
                      enb3 <= 0;
                  end
              s4 :
                 begin
                      st <= s4;
                      X2 <= m5;
                      rdy <= 1'b1;
                 end
              default :
                 begin
                   st <= s0;
                   rdy <= 1'b0;
                 end
           endcase
        end
     end
   end

mult_ip inst(X1, Z2, clk, enb, ready, n1);
modadd  add(X3,Z3,m5);
squar pingfang1(Z3,m1);
squar pingfang2(Z3,m2);
squar pingfang3(Z3,m3);
squar pingfang4(Z3,m4);
endmodule
```

在程序 12-4 中，共 4 个状态循环，得到最终结果。利用一个 always 语句，通过对 st 值进行设置，达到输出 2 倍点结果的目的。

程序 12-5：PXY 模块

```
always @(posedge clk)
  begin
    if (en == 1'b0)
      begin
        X3 <= {233{1'b0}};
          Z3 <= {233{1'b0}};
          X4 <= {233{1'b0}};
```

```
        Z4 <= {233{1'b0}};
      T1 = {233{1'b0}};
      T2 = {233{1'b0}};
      T3 = {233{1'b0}};
      T4 = {233{1'b0}};
      st <= s0;
      rdy <= 1'b0;
      enb1 <= 1'b0;
        enb2 <= 1'b0;
        enb3 <= 1'b0;
        enb4 <= 1'b0;
        enb5 <= 1'b0;
        enb6 <= 1'b0;
        enb7 <= 1'b0;
        enb8 <= 1'b0;
    end
  else  begin
      case (st)
         s0 :
          if(Z1==0)
           begin
              X4 <= 0;
              Z4 <= 0;
           end
            else if(Z2==0)
           begin
              X4 <= x0;
              Z4 <= x0^y0;
           end
             else
               st <= s1;
         s1 :
                 begin
              st <= s2;
              T3 <= x0;
                 T4 <= y0;
                   end
         s2 :
             if(ready1 == 1)
              begin
               T1 <= m1;
                  Z3 <= m2;
               st <= s3;
                   enb1 <= 0;
```

```
        end
            else
              begin
               enb1 <= 1;
           st <= s2;
        end
    s3 :
      if(ready2 == 1)
        begin
         Z4 <= m3;
        st <= s4;
         enb2 <= 0;
        end
            else
              begin
               enb2 <= 1;
           st <= s3;
        end
    s4 :
      begin
            st <= s5;
            Z3 <= n1;
      end
    s5 :
        if(ready3 == 1)
          begin
          X3 <= m4;
          st <= s6;
             enb3 <= 0;
         end
         else
            begin
               enb3 <= 1;
          st <= s5;
         end
       s6 :
            begin
           st <= s7;
           Z4 <= n2;
         end
     s7 :
        if(ready4 == 1)
       begin
          st <= s8;
```

```
        Z4 <= m5;
            enb4 <= 0;
     end
      else
         begin
            enb4 <= 1;
       st <= s7;
      end
s8 :
          begin
        st <= s9;
         T2 <= n3;
     end
 s9 :
begin
       st <= s10;
         T2 <= n4;
end
 s10 :
    if(ready5 == 1)
           begin
      T2 <= m6;
          T1 <= m7;
      st <= s11;
          enb5 <= 0;
 end
        else
          begin
          enb5 <= 1;
     st <= s10;
  end
 s11 :
          begin
       st <= s12;
       T2 <= n5;
  end
 s12 :
  if(ready6 == 1)
           begin
       st <= s13;
       T1 <= m8;
           enb6 <= 0;
    end
         else
```

```
                begin
                enb6 <= 1;
            st <= s12;
        end
       s13 :
      if(ready7 == 1)
               begin
            st <= s14;
            T2 <= m9;
                X4 <= m10;
                enb7 <= 0;
        end
             else
               begin
                enb7 <= 1;
           st <= s13;
        end
       s14 :
            begin
            st <= s15;
            Z4 <= n6;
        end
       s15 :
       if(ready8 == 1)
              begin
            st <= s16;
            Z4 <= m11;
                enb8 <= 0;
        end
            else
              begin
               enb8 <= 1;
            st <= s15;
           end
        s16 :
              begin
            st <= s16;
            Z4 <= n7;
                rdy <= 1;
           end
    default :
      begin
        st <= s0;
        rdy <= 1'b0;
```

```
                end
            endcase
        end
end
mult_ip inst1(Z1, Z2, clk, enb1, ready1, m1);
mult_ip inst2(Z1, T3, clk, enb1, ready1, m2);
mult_ip inst3(Z2, T3, clk, enb2, ready2, m3);
mult_ip inst4(Z4, X1, clk, enb3, ready3, m4);
mult_ip inst5(Z4, Z3, clk, enb4, ready4, m5);
mult_ip inst6(T2, T1, clk, enb5, ready5, m6);
mult_ip inst7(T1, T3, clk, enb5, ready5, m7);
inverse   fan(T1, clk, enb6, ready6, m8);
mult_ip inst8(T2, T1, clk, enb7, ready7, m9);
mult_ip inst9(T1, X3, clk, enb7, ready7, m10);
mult_ip inst10(Z4, T2, clk, enb8, ready8, m11);
MOD_ADD  jia1(Z3, X1,n1);
MOD_ADD  jia2(Z4, X2,n2);
squar pingfang1(T3,n3);
MOD_ADD  jia3(T2, T4,n4);
MOD_ADD  jia4(T2, Z4,n5);
MOD_ADD  jia5(X4, T3,n6);
MOD_ADD  jia6(Z4, T4,n7);
endmodule
```

程序 12-6：PointMult

```
always @(posedge clk)
  begin
      if (en == 1'b0)
         begin
            X1 <= m3;
            Z1 <= 1;
              X2 <= m2 + 1'b1;
            Z2 <= m1;
              cnt <= m-2;
            st <= s0;
            rdy <= 1'b0;
              enb1 <=0;
              enb2 <=0;
              enb3 <=0;
          end
       else
         begin
           case (st)
             s0 :
             if(cnt <= 0)
```

```
                begin
                    st <= s0;
                     cnt <= cnt - 1'b1;
                  if(k[cnt]==1)
                   begin
                    if(ready1 == 1)
                        begin
                         X1 <= m3;
                       Z1 <= m4;
                         X2 <= m5;
                       Z2 <= m6;
                         enb1 <= 0;
                        end
                     else
                        enb1 <= 1;
                    end
                  else
                   begin
                    if(ready2 == 1)
                        begin
                         X1 <= m9;
                       Z1 <= m10;
                         X2 <= m7;
                       Z2 <= m8;
                         enb2 <= 0;
                        end
                     else
                        enb2 <= 1;
                    end
                end
              else
                st <= s1;
              s1 :
                if(ready3 == 1)
                    begin
                        x3 <= m11;
                     y3 <= m12;
                         rdy <= 1'b1;
                         st <= s1;
                        enb3 <= 0;
                       end
                     else
                        enb3 <= 1;
                         st <= s1;
                    end
                default :
                  begin
```

```
                    st <= s0;
                    rdy <= 1'b0;
                    end
            endcase
        end
    end
    squar pingfang1(x0,m1);
    squar pingfang2(m1,m2);
    PADD adder1(x0,X1,Z1,X2,Z2, clk, enb1, ready1, m3,m4);
    PADD adder2(x0,X2,Z2,X1,Z1, clk, enb2, ready2, m7,m8);
    PDOUBLE  adder3(X2,Z2, clk, enb1, ready1,m5,m6);
    PDOUBLE  adder4(X1,Z1, clk, enb2, ready2,m9,m10);
    PXY   (x0, y0,X1,Z1,X2,Z2,clk, enb3, ready3, m11,m12);
    endmodule
```

在程序 12-6 中，采用 case 语句进行循环操作，st 共有两个状态：其中 s0 状态实现 cnt 从 m–2 到 0 的循环，在这个循环中，若 k[cnt]=1，调用 $(X_1,Z_1)\leftarrow \mathrm{PADD}(X_1,Z_1,X_2,Z_2)$, $(X_2,Z_2)\leftarrow \mathrm{PDOUBLE}(X_2,Z_2)$ 模块，若 k[cnt]=0，调用 $(X_2,Z_2)\leftarrow \mathrm{PADD}(X_2,Z_2,X_1,Z_1)$, $(X_1,Z_1)\leftarrow \mathrm{PDOUBLE}(X_1,Z_1)$ 模块；在 s1 状态，调用坐标转换模块 $Q=\mathrm{PXY}(X_1,Z_1,X_2,Z_2)$，并输出最终结果。

12.3　ECC 数字签名算法的工程实现及结果

1．创建 ISE 工程

打开 ISE 开发环境，选择菜单 File→New Project，建立一个新工程，然后设置顶层文件模块名、存储目录和设计方式。接着选择器件，这里选择 Xilinx 的 Spartan6 来实现。最后再选择第三方仿真软件 ModelSim 和 Verilog 语言进行仿真测试。新工程的建立如图 12.3 和图 12.4 所示。

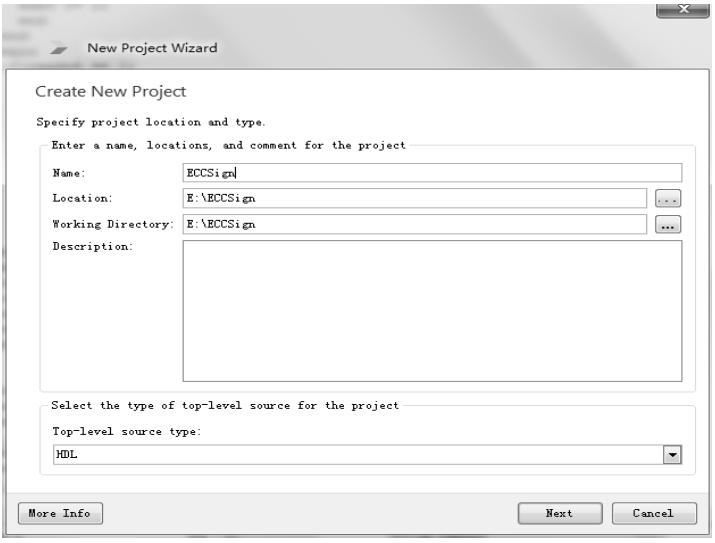

图 12.3　ECC 数字签名算法工程文件的建立

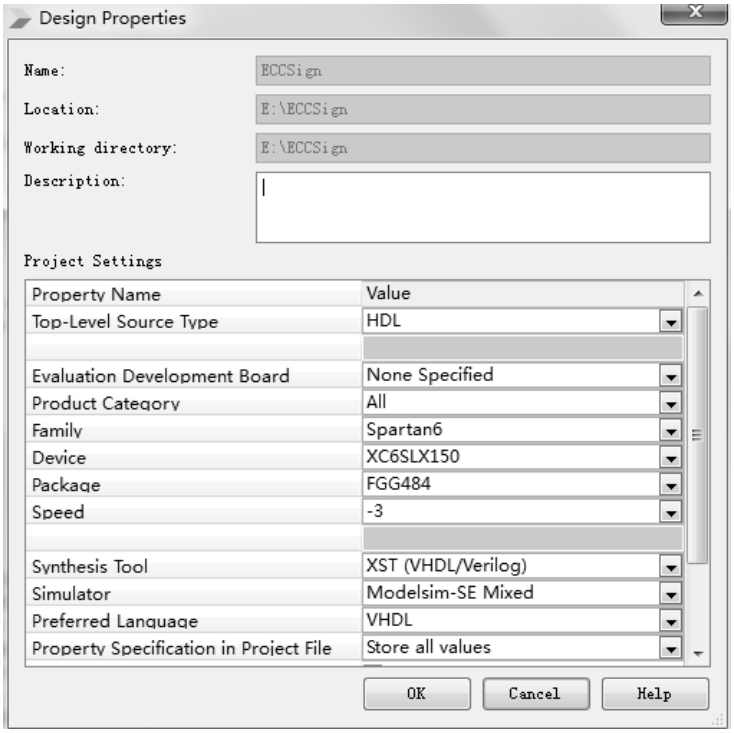

图 12.4 器件的选择和工具的使用

2．编写设计代码

在 12.2 节中详细叙述了 ECC 数字签名算法的关键模块设计，上面的模块就不做赘述。以下是在算法中用到的模加模块的程序代码。

程序 12-7：

```
module MOD_ADD(
                DIN1,
                DIN2,
                DOUT
                );
input [232 : 0] DIN1;
input [232 : 0] DIN2;
output [232 : 0] DOUT;
assign   DOUT = DIN1 ^ DIN2;
endmodule
```

在程序 12-7 中，由于二进制域的加法相对简单，故该运算模块用异或门即可实现。

3．工程验证

（1）根据以上编写的程序，编写该程序的测试文件，如图 12.5 和图 12.6 所示。

（2）根据上一步的测试文件，调用 ModelSim 仿真软件，如图 12.7 所示，对该算法进行仿真测试，仿真结果如图 12.8 所示。

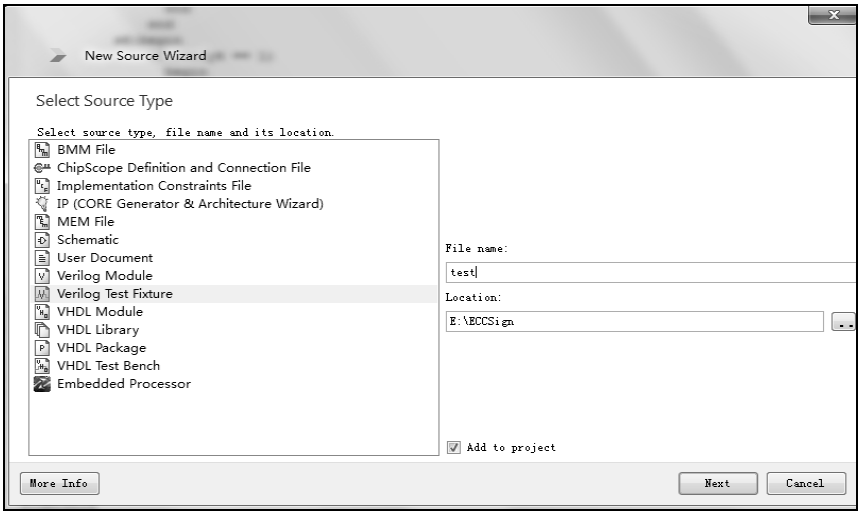

图 12.5　ECC 数字签名算法测试文件的建立

图 12.6　ECC 数字签名算法测试文件

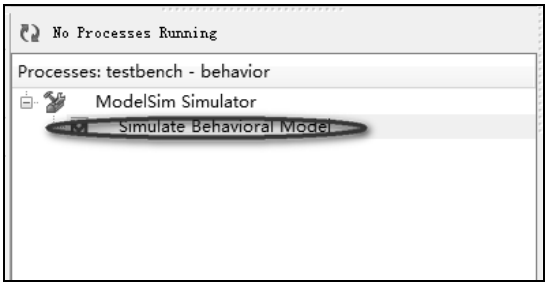

图 12.7　ModelSim 调用示意图

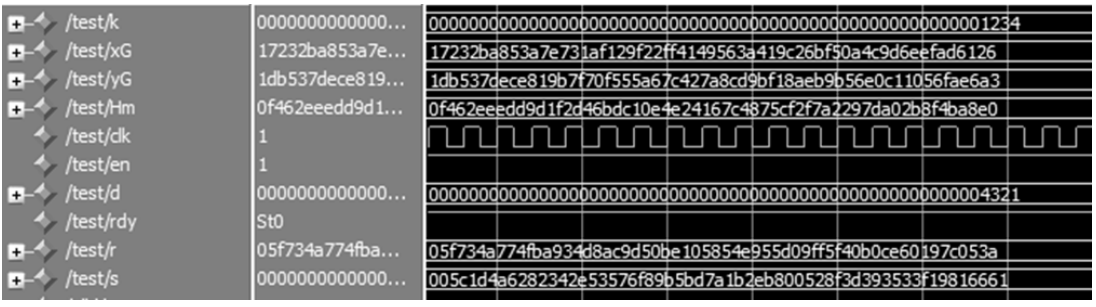

图 12.8　ECC 数字签名算法仿真结果

12.4 效果测试

通过功能仿真与系统测试，可以验证整个设计的正确性，为了测试其性能，还需要对整个设计在芯片上的综合结果进行分析。

仿真时，在波形文件中需要给出合适的时钟周期。在 FPGA 程序设计中，验证算法正确性和设计合理性的方法之一是对程序进行波形仿真。在本实例中使用 ModelSim 对算法进行仿真，以验证其功能是否正确。

1．点乘运算仿真结果（如图 12.9 所示）

选取参数如下：

x0=233'h17232BA853A7E731AF129F22FF4149563A419C26BF50A4C9D6EEFAD6126;

y0=233'h1DB537DECE819B7F70F555A67C427A8CD9BF18AEB9B56E0C11056FAE6A;

k = 233'h1234。

结果为：

x=233'h0AB11A49A966B6199E115C5EF0344E28A1201FB08FD69CD6E4B9637ADDA;

y=233'h06FE3F4A27C0C422BE77B910F5D992219A8FEC487FE40431FBC0BFD822E。

图 12.9　ECC 数字签名算法点乘运算的仿真结果

2．ECC 数字签名仿真结果

选取参数如下：

xG = 233'h17232BA853A7E731AF129F22FF4149563A419C26BF50A4C9D6EEFAD6126;

yG = 233'h1DB537DECE819B7F70F555A67C427A8CD9BF18AEB9B56E0C11056FAE6A3;

Hm = 233'h0f462eeedd9d1f2d46bdc10e4e24167c4875cf2f7a2297da02b8f4ba8e0;

d = 233'h4321;

k = 13'h1234。

结果为：

r = 233'h05f734a774fba934d8ac9d50be105854e955d09ff5f40b0ce60197c053a;

s = 233'h005c1d4a6282342e53576f89b5bd7a1b2eb800528f3d393533f19816661。

下面对该工程的时序、运行效率、资源占用情况进行分析。在 Xilinx 公司的 6slx150fgg484 FPGA 元器件上，经过 ISE 的 XST 分析工具进行综合。综合结果如图 12.10 所示。

资源占用情况分析报告如表 12.1 所示。

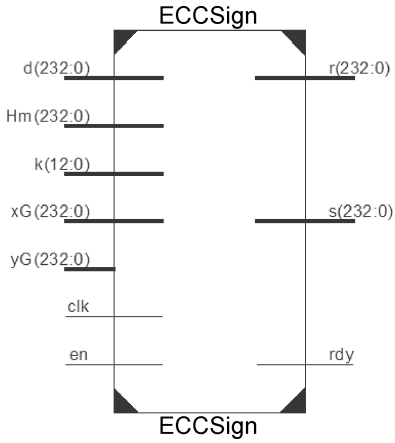

图 12.10　ECC 数字签名算法综合后的外部器件

表 12.1　ECC 数字签名算法资源占用情况

Selected Device : 6slx150fgg484-3	
Number of Slice Registers	16156　out of　184304　8%
Number of Slice LUTs	74352　out of　92152　80%
Number used as Logic	74352　out of　92152　80%
Number with an unused Flip Flop	61348　out of　77504　79%
Number with an unused LUT	3152　out of　77504　4%
Number of fully used LUT-FF pairs	13004　out of　77504　16%
Number of BUFG/BUFGCTRLs	8　out of　16　50%

时序分析报告如下。

Timing Summary:

Minimum period: 6.395 ns (Maximum Frequency: 156.369 MHz)

Minimum input arrival time before clock: 9.033 ns

Maximum output required time after clock: 5.078 ns

Maximum combinational path delay: No path found

由时序分析报告结果可知，ECC 数字签名算法的最高工作频率为 156.369 MHz，分析程序可知，处理每 233 比特的分组数据需要经过 915 个时钟，所以，ECC 数字签名算法的最高处理速度为 233×156.369/915=39.9 Mbps。

由以上分析报告可以看出，整个工程在 6slx150fgg484-3 芯片上使用的查找表数为 74352，占芯片资源的 80%；使用了 16156 个寄存器（Registers），占引脚资源的 8%。从中可以看出，此工程很好地利用了选取的逻辑元器件资源，达到了较好的运算性能。

12.5　本章小结

本章主要对 ECC 数字签名算法进行了介绍，包括基本原理与 FPGA 硬件实现方法，并分别给出了关键模块的实现代码和仿真结果，最后，对该算法工程进行综合，得出 ECC 数字签名算法的运算效率和资源利用情况，使读者对该算法的 FPGA 实现有了进一步的认识。

参 考 文 献

[1] 褚振勇, 翁木云. FPGA 设计及应用[M]. 西安: 西安电子科技大学出版社, 2002.

[2] 田家林, 陈利学, 寇向辉. FPGA 在运动控制系统中的设计[J]. 微计算机信息, 2007(8).

[3] 余佳. 基于 FPGA 的设计与实现[J]. 计算机与数字工程, 1995(01).

[4] 王晓勇. FPGA 的基本原理及运用[J]. 舰船电子工程, 2005(02).

[5] 王简瑜, 张鲁国. 基于 FPGA 实现 DES 算法的性能分析[J]. 微计算机信息, 2007, 23(8).

[6] Wilson P. FPGA 设计实战[M]. 杜生海，译. 北京: 人民邮电出版社, 2009.

[7] 肖新帅, 刘洪鹏. 基于 FPGA 的 DES 算法的并行加密技术[J]. 科技信息, 2010(25).

[8] 李辉. 基于 FPGA 的数字系统设计[M]. 西安: 西安电子科技大学出版社，2008.

[9] 杨波. 现代密码学[M]. 北京: 清华大学出版社, 2003.

[10] P. Kocher, J. Jaffe, B. Jun. Differential Power Analysis[C]. In: Proceedings of Advances in Cryptology, 1999, volume 1666 of Lecture Notes in Computer Science(LNCS), 388-397.

[11] P. C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS and Related Attacks[C]. In: Proceedings of Advances in Cryptology-CRYPTO, 1996.

[12] Yuval Ishai, Amit Sahai, David Wagner. Private Circuits: Securing Hardware against Probing Attacks. In: Proceedings of Advances in Cryptology(CRYPTO 2003), volume 2729 of Lecture Notes in Computer Science. New York: Springer Verlag, 2003, 463-481.

[13] J. Domingo-Ferrer, “A Provably Secure Additive and Multiplicative Privacy Ho-momorphism, ” ISC2002, LNCS. Vol. 2443, pp. 471-483, 2002.

[14] H. Saputra, N. Vijaykrishnan, M. Kandemir, et al. Masking the Energy Behavior of DES Encryption[C]. In: Proceedings of Design Automation and Test in Europe Conference(DATE 03), Munich, Germany, 2003: 84-89.

[15] Johannes Blomer, Jorge Guajardo Merchan, Volker Krummel. Provably secure masking of AES[C]. In: Proceedings of Selected Areas in Cryptography, 2004.

[16] J. D. Golic, Ch. Tymen. Multiplicative Masking and Power Analysis of AES[C]. In: Proceedings of Cryptographic Hardware and Embedded Systems (CHES 2002). Springer-Verlag, 2003: 198-212.

[17] 张淑芬，郝福珍. RSA 算法在 FPGA 上的实现[J]. 计算机工程与设计, 2010, 31(13): 2962-2965.

[18] 王旭, 董威, 戎蒙恬. 基于改进 Montgomery 模乘算法的 RSA 加密处理器的实现[J]. 上海交通大学学报, 2004, 38(02): 240-243.

[19] G. Feng, G. S. Ma, Z. Yang. Implementation of RSA Based on Modified Montgomery Modular Multiplication Algorithm. International Conference on Scientific Computing. 2006 .

[20] 寇文. RSA 密码芯片的 FPGA 实现[D]. 郑州：解放军信息工程大学工学硕士学位论文，2005: 17-20.

[21] 吴武飞, 王奕, 李仁发. 可重构 Keccak 算法设计及 FPGA 实现[J]. 计算机应用，2012，32(3) : 864-866.

[22] 张花, 崔慧娟, 唐昆. 一种 RSA 算法之数字签名系统的快速实现方案[J]. 计算机工程, 2006, 32(03): 156-157.

[23] 周玉洁, 冯登国. 公开密钥密码算法及其快速实现[M]. 北京：国防工业出版社, 2002: 88-100.

[24] 王张宜, 杨寒涛, 张焕国. 椭圆曲线密码的安全性分析[J]. 计算机工程, 2002, 28(5): 161-163.

[25] 白国强, 陈弘毅. 椭圆曲线密码的芯片集成[J]. 中兴通讯技术, 2004, 4: 27-29.

[26] 胡端元, 陈文宇, 甘骏人, 等. 椭圆曲线加密的硬件实现[J]. 电子设计应用, 2003. 5.

[27] Agnew G B, Mullin R C, Vanstone SA. An Implementation of Elliptic Curve Cryptosystem over) [J]. IEEE J on Selected Areas in Communications 1993, 11(5): 804-813.

[28] 冯登国, 裴定一. 密码学导论[M]. 北京：科学出版社, 1999.

[29] 夏宇闻. 复杂数字电路与系统的 verilog HDL 设计技术[M]. 北京：北京航空航天大学出版社, 1998.

[30] 王学理, 斐定一. 椭圆与超椭圆曲线公钥密码的理论与实现[M]. 北京: 科学出版社, 2006.

[31] 国家密码管理局. SM2 椭圆曲线公钥密码算法. http: //www. oscca. gov. cn/UpFile/2010122214822692. pdf, 2010.

[32] 国家密码管理局. SM2 椭圆曲线公钥密码算法推荐曲线参数. http: //www. oscca. gov. cn/UpFile/2010122214836668. pdf, 2010.

[33] 孟德欣, 俞国亮. SHA-1 算法的 HDL 设计与仿真[J]. 计算机仿真, 2009(6).

[34] 孙黎, 慕德俊, 刘航. 基于 FPGA 的 SHA-1 算法的设计与实现[J]. 计算机工程, 2007(14).

[35] 龚源泉, 沈海斌, 何乐年, 等. SHA-1 加密算法的硬件设计[J]. 计算机与工程应用，2004, 40(3).

[36] 章照止. 现代密码学基础[M]. 北京: 北京邮电大学出版社, 2004.

[37] 于工, 牛秋娜, 朱习军, 等. 现代密码学原理与实践[M]. 西安: 西安电子科技大学出版社, 2009.1.

[38] Bruce Schneier. 应用密码学: 协议算法与 C 源程序[M]. 北京: 机械工业出版社, 2000.1.

[39] 李长可. 基于 FPGA 可重构快速密码芯片设计[J]. 计算机测量.

[40] 杨宏志, 韩文报, 董博. 类 AES 分组密码统一框架及其 FPGA 实现［J］. 计算机科学，2010.

[41] 国家密码管理局. SM3 密码杂凑算法[EB/OL]. (2010-12-22). http: //www. oscca. gov. cn/UpFile/201012221418577866. pdf.

[42] 杨晓辉, 戴紫彬. 基于 FPGA 的 SHA-256 算法实现[J]. 微计算机信息, 2006, 22(11): 146-148.

[43] 刘宗斌, 马原, 荆继武, 等. SM3 哈希算法的硬件实现与研究[J]. 信息网络安全, 2011(09).

[44] HOMSIRIKAMOL E, ROGAWSKI M, GAJ K. Comparing hard-ware performance of fourteen round two SHA-3 candidates using FPGAs. http: //eprint. iacr. org/2010/445. pdf . 2010.

[45] 董馨. 可配置 SHA-2 系列算法和 SHA-3（BLAKE-32）算法的硬件实现[D]. 华中科技大学, 2012.

[46] 田茂松. 基于椭圆曲线密码体制的数字签名算法及其 FPGA 实现[D]. 武汉理工大学, 2007.

[47] 王晓花, 赵耿. 基域 GF(2^{233})上椭圆曲线密码体制（ECC）在智能卡上的实现[A]. //2006 北京地区高校研究生学术交流会——通信与信息技术会议论文集（下）[C]. 2006.

[48] 祁明, 肖国镇. 数字签名算法的改进[J]. 电子学报, 1997(04).

[49] R L Rivest, A Shamir, L Adleman. A method for obtaining digital signatures and public key cryptosystems[J]. Communications of the ACM, 1978, 21(2): 120-126.

[50] J Gordon, Strong Primes are Easy to Find Eurocrypt. 1984: 216-223.

[51] JJ Quisquater, C Couvreur. Fast DeciPherment Algorithm for RSA Public-K e y Cryptosystem: Electronics Letters. 1982, 18(21): 905-907.

[52] AOL Atkin, RG Larson. On a Primality. Test of Solovay and Strassen. SIAM J. COMPUT. 1982, 11(4): 789-791.

[53] P Paillier. Public-Key cryptosystems based on composite degree residuosity classer. In: Wiener MJ, ed. Proc. Of the EUROCRYPT'99. LNCS 1592, Springer-Verlag, 1999: 223-238.

[54] D Naccache, J Stern. A new public key cryptosystem based on higher residues. In: Proc. Of the 5th Symp. On Computer and Communications Security. ACM Press, 1988: 59-66.

[55] Galbraith SD. Elliptic curve Paillier schemes. Journal of Cryptology, 2002, 15(2): 129-138.

[56] D Catalano, R Gennaro, N Howgrave-Graham, PQ Nguyen. Paillier's cryptosystem revisited. In: Proc . of the 8th ACM Conf. on Computer and Communication Security. ACM Press, 2001: 206-214.

[57] I Damgard, M Jurik M. A generalization, a simplification and some applications of Paillier's probabilistic public-key system. In: K Kim, ed. Proc. of the PKC 2001. LNCS 1992, Springer-Verlag, 2001: 119-136.

[58] J. Hoffstein, J. Pipher, J. H. Silverman. NTRU: A ring-based public key cryptosystem[C]. In: Algorithmic Number Theory (ANTS-III), LNCS 1423, Berlin: Springer-Verlag, June 1998: 267-288.

[59] C Gentry. Key recovery and message attacks on NTRU-composite[C]. Advances in Cryptology-Eurocrypt 2001, LNCS 2045, Springer-Verlag, 2001: 182-194.

[60] Nick Howgrave-Graham. A Hybrid Lattice-Reduction and Meet-in-the- Middle Attack against NTRU. Advances in Cryptology-Crypto 2007, LNCS 4622, Springer-Verlag, 2007: 150-169.

[61] J. Hoffstein，J. Silverman，Optimizations for NTRU，in Public-Key Cryptography and Computational Number Theory(Warsaw, September 11-15，2000).

[62] C. Gentry, J. Jonsson，J. Stern, M. Szydlo，Cryptanalysis of the NTRU Signature SCheme(NSS) fromEurocrypt'01，Advances in Crytology-Asilcrypt'01，LNCS，V01. 2348，Springer-Verlag，2001，PP. 123-131.

[63] C. Gertry, M. Szydlo，Cryptanalysis of the Revised NTRU Signature Scheme，Advances in Cryptology。Eurocrypt'02，LNCS，V01. 2332, Springer-Vetag，2002，PP. 299-320.

[64] S. Min，G Yamamoto，Weak property of malleability in NTRUSign，ACISP04；July 13-15 ，Sydney, Australia.

[65] W. Diffie, M. E. Hellman. New Directions in Cryptography, IEEE Transactions on Information Theory, v. IT-22, n. 6. Nov 1976, pp. 644-654.

[66] A. Miyaji. A Message recovery Signature Scheme Equivalent to DSA over Elliptic Curves, in Advances in Cryptology-ASIACRYPT'96, Lecture Notes in Computer Science, 1163, Berlin: Springer-Verlag, 1997, pp. 1-14.

[67] K. Nyberg, R. A. Rueppel, Message recovery for signature schemes based on the discrete logarithm problem, Advances in Cryptology-Proceedings of Eurocrypt'94. Lecture Notes in Computer Science, 950(1995), Springer-Verlag, pp. 182-193.

[68] MJ Potgieter, BJ van Dyk. Two Hardware Implementations of the Group operations Necessary for Implementing an Elliptic Curve Cryptosystem over a Characteristic Two Finite Field[J]. IEEE Africon 2002. pp: 187-192.

[69] ANSI-X9. 63-1998: Public Key Cryptography for Financial Service Industry: Key Agreement and Key Transport Using Elliptic Curve Cryptography[S]. American Bankers Association，August 1998.

[70] IEEE P1363, Standard Specifications for Public Key Cryptography Draft Version 13[S]. 1999.

[71] McEliece, Robert J. A Public-key Cryptosystem Based on Algebraic Coding Theory. Technical Report, Jet Propulsion Lab. DSN Progress Report, 1978: 42-44.

[72] Bernstein，Daniel J，Buchmann，Johannes A. Post-Quantum Cryptography. Springer, Heidelberg, 2009.

[73] Catterall N, Gabidulin E M, Honary B, Obemikhin V A. Public Key Cryptosystem based metrics associated with GRS Codes. ISIT2006, 2006: 729-733.

[74] Sidelnikov V M, Shetakov S O. On the insecurity of Cryptosystem Based on Generalized Reed-Solomon Codes. Discrete Math, 1922, 2(4): 439-444.

[75] C Wieschebrink. An attack on a modified Niederreiter encryption scheme [A]. In Proceedings of the 9th International Conference on Theory and Practice of Public-Key Cryptography [C]. New York, NY, USA, 2006, (3958): 14-26.

[76] Wang X Y, Lai X J, Feng D G, et al. Cryptanalysis of the hash functionsMD4 and IPEMD[C]// ramer (ed). EUROCRYPT 2005, LNCS3494. Berlin: Springer-Verlag, 2005: 1-18.

[77] Wang X Y, Yu H B, Yiqun Lisa Yin. Efficient collision search attacks on SHA-0[C]//Shoup (ed). CRYPTO 2005, LNCS 3621. Berlin: Springer-Verlag, 2005: 1-16.

[78] Wang X Y, Yiqun Lisa Yin, Yu H B. Finding collisions in the full SHA-1[C]//Shoup (ed). CRYPTO 2005, LNCS 3621. Berlin: Springer-Verlag, 2005: 17-36.

[79] Bozhan Su, Wenling Wu, Shuang Wu, Le Dong. Near-Collisions on the Reduced-Round Compression Functions of Skein and BLAKE. Cryptology and Network Security, Volume 6467, pp. 124-139, 2010.

[80] Praveen Gauravaram, Lars R. Knudsen，et al. Groestl-a SHA-3 candidate[EB/OL]. http: //www. Groestl. info.

[81] Hongjun Wu. The Hash Function JH[EB/OL]. http: //www3. ntu. edu. sg/home/wuhj/research/ jh/jh_round3. pdf.

[82] Maurer U, Renner R, Holenstein C. Indifferentiability, impossibility results on reductions, and applications to the random oracle methodology[C]//Naor (ed). TCC'04, LNCS 2951. Berlin: Springer-Verlag, 2004: 21-39.

[83] Guido Bertoni, Joan Daemen, et al. Keccak specifications [EB/OL]. http: //keccak. noekeon. org/.